Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Herr Dipl.-Informatiker Frank Huber

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Elmar Kulke

Gutachter:
1. Prof. Johann-Christoph Freytag, Ph.D.
2. Prof. Dr. Volker Markl
3. Prof. Dr.-Ing. Wolfgang Lehner

eingereicht am: 19. Mai 2011
Tag der mündlichen Prüfung: 17. Februar 2012

Abstract

The upcoming generation of many-core architectures poses several new challenges for software development: Software design and software implementation has to change from sequential execution to a highly parallel execution, such that it takes full advantage of the steadily growing number of cores on a single processor.

With this thesis, we investigate such highly parallel program execution in the context of relational database management systems (RDBMSs). We consider the complete process of query processing and identify four problem areas which are crucial for efficient parallel query processing on many-core architectures. These four areas are: Hardware, physical data model, query execution, and query optimization. Furthermore, we present a framework which covers all four parts, one after another.

First, we give a detailed survey of computer hardware with a special focus on memory and processors. Based on this survey we propose a hardware model. Our abstraction aims to simplify the task of software development on many-core hardware. Based on the hardware model, we investigate physical data models and evaluate how the physical data model may support optimal query execution by providing efficient and parallelizable data structures. Additionally, we design a new index structure that utilizes data parallel execution by using SIMD operations. The next layer within our framework is query execution, for which we present a new task based query execution model. Our query execution model allows for a lightweight parallelism. Finally, we cover query optimization by explaining approaches for optimizing resource utilization on a query local point of view as well as query global point of view.

Zusammenfassung

Der Trend zu immer mehr parallelen Recheneinheiten (Kernen) innerhalb eines einzelnen Prozessors stellt an die Softwareentwicklung neue Herausforderungen. Um die vorhandenen Ressourcen vollständig auszulasten und die stetige Steigerung der Parallelität – durch die Erhöhung der Anzahl der Kerne von einer Prozessorgeneration zur Nächsten – in einen Leistungszuwachs umzusetzen, muss Software von der sequentiellen Verarbeitung in eine hochgradig parallele Verarbeitung übergehen.

Diese Arbeit untersucht, wie solch eine hoch parallele Verarbeitung in Bezug auf Relationale Datenbankmanagementsysteme (RDBMSe) umzusetzen ist. Dazu wird zunächst der gesamte Prozess der Anfragebearbeitung betrachtet und vier Problembereiche identifiziert, die für das Ziel der parallelen Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen maßgeblich sind. Diese Bereiche sind die Hardware selbst, das physische Datenmodell sowie die Anfrageausführung und -optimierung. Diese vier Bereiche werden schrittweise innerhalb eines Rahmenwerkes betrachtet.

Nach einer kurzen Einführung, wird sich die vorliegende Arbeit zunächst mit Grundlagen befassen. Dazu werden die Hardwarebestandteile Speicher und Prozessor detailliert betrachtet und ihre Funktionsweise und Eigenschaften erläutert. Auf diesem Wissen aufbauend, werden ein Hardwaremodell und seine Operatoren definiert. Sie ermöglichen eine von der jeweiligen Hardwarearchitektur unabhängige Softwareentwicklung, ohne den Verlust an Funktionalität und Leistung durch die jeweiligen Besonderheiten der Hardwarekomponenten. Im Weiteren wird das physische Datenmodell untersucht und analysiert, wie das physische Datenmodell eine optimale Anfrageausführung unterstützen kann. Die verwendeten Datenstrukturen müssen dafür einen effizienten und parallelen Zugriff erlauben. Die Analyse führt zur Entwicklung eines neuartigen Indexes, der die datenparallele Abarbeitung (SIMD) nutzt. Gefolgt wird dieser Teil von der Anfrageausführung, in der ein neues Anfrageausführungsmodell entwickelt wird, das auf der Verwendung des Taskkonzepts beruht und eine hohe und sehr leicht gewichtige Parallelität erlaubt. Den Abschluss stellt die Anfrageoptimierung dar, worin verschiedene Ideen für die Optimierung der Ressourcenverwaltung präsentiert werden.

Ich widme diese Arbeit
meiner Familie und meinen Freunden

Danksagung

Ich möchte mich an dieser Stelle bei allen bedanken, die mich auf dem Weg dieser Arbeit begleitet und unterstützt haben.

Zuerst möchte ich Herrn Prof. J.-C. Freytag danken, dass er mir die Möglichkeit gegeben hat, am DBIS Lehrstuhl diese Arbeit zu schreiben und mich mit vielen hilfreichen Kommentaren und Diskussionen immer wieder unterstützt hat. Ebenfalls möchte ich mich für die mir zuteil gewordenen Freiheiten und das Vertrauen bedanken, das er mir in den sieben Jahren entgegengebracht hat, sowie für seine Anteilnahme und seine aufbauenden Worte in Zeiten meiner persönlicher Trauer.

Ich möchte meiner Kollegin Katja Tham, meinen Kollegen Olaf Hartig und Ralf Heese für ihre kritischen Bemerkungen und konstruktiven Gespräche danken. Weiterhin danke ich Frau Ulrike Scholz, Herrn Dr. Thomas Morgenstern sowie Herrn Heinz Werner für ihre administrative und technische Unterstützung.

Allen Freunden, die mich über die Jahre auf unterschiedlichste Art und Weise unterstützt und begleitet haben, gilt ebenfalls mein Dank.

Bei meinen Eltern möchte ich mich für ihre Hilfe und die Unterstützung über die gesamte Zeit des Studiums und der Dissertation bedanken. Ihre Unterstützung hat mir die Konzentration auf die Arbeit zutiefst erleichtert.

Bei meiner Frau Ilona bedanke ich mich für die Liebe, die sie mir entgegenbringt und die Freiräume, die sie mir gewährt hat.

Am Ende danke ich meinen Töchtern Johanna und Helen für ihre unendliche Liebe, das Glück und den Spaß, den sie in mein Leben bringen.

Inhaltsverzeichnis

1 Einleitung

"'I know how to make 4 horses pull a cart - I don't know how to make 1024 chickens do it."'

Enrico Clementi (1931–)

1.1 Motivation

Die Computerwelt befindet sich seit einiger Zeit in einem grundlegenden, technologischen Wandel. Sie vollzieht dabei den Wechsel von einer seriellen hin zu einer hoch parallelen Informationsverarbeitung.

Transistoren bilden, seit ihrer Entwicklung Ende der vierziger Jahre, eine enorm wichtige Grundlage für Computer. Ihre Fähigkeit Strom in Abhängigkeit vom Eingangssignal fließen zu lassen, wird genutzt, um komplexe Schaltungen zu erstellen. Dabei werden viele Millionen gar Milliarden von Transistoren auf integrierten Schaltkreisen (ICs) untergebracht.

IC – Integrated Circuit

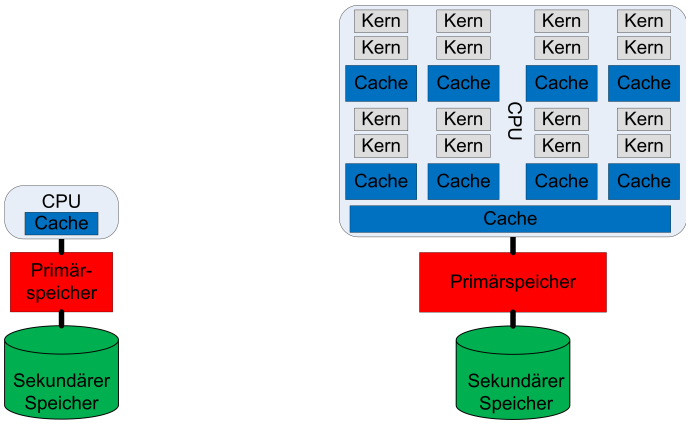
Das Mooresche Gesetz [126] beschreibt die Entwicklung von Transistoren auf solchen ICs. Es besagt, dass sich etwa alle 18 bis 24 Monate die Anzahl der Transistoren auf einem IC verdoppeln lässt. Dieser Effekt führte in der Vergangenheit zu immer besseren bzw. schnelleren Prozessoren und Computern. Er führte auch automatisch zu einer stetigen Verbesserung des Leistungsverhaltens von Software. Während das Gesetz von Moore weiterhin Gültigkeit hat, sind einer weiteren Steigerung der Leistung von Prozessoren bestimmte physikalische und wirtschaftliche Grenzen gesetzt. Diese Grenzen haben zu einem neuen Trend bei der Entwicklung von Prozessoren geführt, den Mehrkern-Prozessoren.

Bei Mehrkern-Prozessoren werden die zusätzlichen Transistoren nicht mehr für die Leistungssteigerung eines einzelnen Prozessors verwendet [118]. Stattdessen werden mehrere homogene oder auch heterogene Recheneinheiten, sogenannte **Kerne**, im gleichen Prozessorgehäuse untergebracht. Dieser Trend wird in naher Zukunft zu Prozessoren mit einer hohen Anzahl von Kernen führen [159]. Damit kann ein einzelner Prozessor eine Vielzahl von Prozessen, Threads bzw. Tasks gleichzeitig ausführen.

Das Mooresche Gesetz hat aber auch zu einem Missverhältnis der Zugriffszeiten in der Speicherhierarchie geführt. Zugriffe auf Daten, die nicht in den prozessornahen Caches liegen, benötigen bis zu mehrere hundert Prozessorzyklen. Diese Latenz in den Zugriffs-

zeiten kann von der CPU meist nur durch Warten überbrückt werden. In diesen Fällen geht wertvolle Rechenzeit durch das Warten der CPU verloren, was sich letztendlich in einem schlechten Leistungsverhalten bemerkbar macht.

Eine weitere Konsequenz des Mooreschen Gesetzes ist die stetig größer werdende Hauptspeichergröße bei abnehmendem Preis. Damit sind große Hauptspeicherinstallationen im Gigabyte Bereich und zunehmend auch im Terabyte Bereich keine Seltenheit mehr. Dieses führt wiederum dazu, dass mehr und mehr Daten bereits im Speicher stehen und nicht mehr von der Festplatte gelesen werden müssen. Aus diesem Grund wird der Datenaustausch zwischen Hauptspeicher und Prozessor immer mehr zu einer beschränkenden Größe.



(a) Einzelkern

(b) Mehrkern

Abbildung 1.1: Wandel der Rechnerarchitektur

Die Folgen des Mooreschen Gesetzes treffen natürlich auch **Relationale Datenbankmanagementsysteme** (RDBMSe). Im Zuge der Mehrkern-Prozessoren müssen sich auch RDBMSe an den Wechsel von der alten Einzelkern-Rechnerarchitektur (Abb. 1.1(a)) hin zur neuen Mehrkern-Rechnerarchitektur (Mk-RA, Abb. 1.1(b)) anpassen. Die großen Hauptspeicherinstallationen erlauben es eine Vielzahl an Daten im Hauptspeicher zu puffern und nicht erst beim eigentlichen Zugriff von der Festplatte lesen zu müssen. Heutige RDBMSe sind oft noch auf die Optimierung der Zugriffe auf den Sekundärspeicher – Festplatten – fokussiert, hier muss ein Umdenken in der Architektur und Softwareentwicklung von RDBMSen stattfinden. Künftige RDBMSe werden sich mit einem enormen Grad an Parallelität und der Optimierung von Hauptspeicherzugriffen aus der CPU beschäftigen müssen.

RDBMSe werden heutzutage in vielen verschiedenen Anwendungsbereichen und für viele verschiedene Zwecke verwendet. Dabei können die Anfragen und ihre Aufgaben von sehr einfach bis hoch komplex variieren. Die Daten, die für die einzelnen Aufgaben verwendet werden, können von wenigen Bytes bis zum Tera- und Petabyte-Bereich reichen.

1.1.1 Data-Warehouse

Ein wichtiges Anwendungsgebiet von RDBMSen stellt der **Data-Warehouse-Bereich** dar. Inmon [89, 90] definiert ein Data-Warehouse wie folgt:

> "'The data warehouse is a basis for informational processing. It is defined as being: subject oriented; integrated; nonvolatile; time variant; a collection of data in support of management's decision."'

Definition 1.1 (Data-Warehouse) Demnach stellt ein Data-Warehouse eine Basis zur Informationsverarbeitung dar, die die folgenden Eigenschaften hat:

- **Subjekt orientiert** – Die Daten geben Informationen über bestimmte Subjekte anstatt Informationen über den innerbetrieblichen Ablauf.
- **Integriert** – Daten werden aus verschiedenen Quellen extrahiert und im Data-Warehouse zusammengefügt.
- **Zeit-variant** – Alle Daten im Data-Warehouse können einem bestimmten Zeitraum zugeordnet werden.
- **Nicht volatile** – Daten in einem Data-Warehouse sind stabil; das bedeutet, dass zusätzliche Daten hinzugefügt werden können, aber Daten niemals gelöscht werden. □

Ein Data-Warehouse ist ein Datenbanksystem, dessen Daten sich aus Daten verschiedener integrierter Datenquellen zusammensetzt (s. Abb. 1.2(a)). Meist handelt es sich bei diesen Datenquellen um Datenbanken, auf denen im normalen Tagesgeschäft – mit kurzen Transaktionen – gearbeitet wird (OLTP).

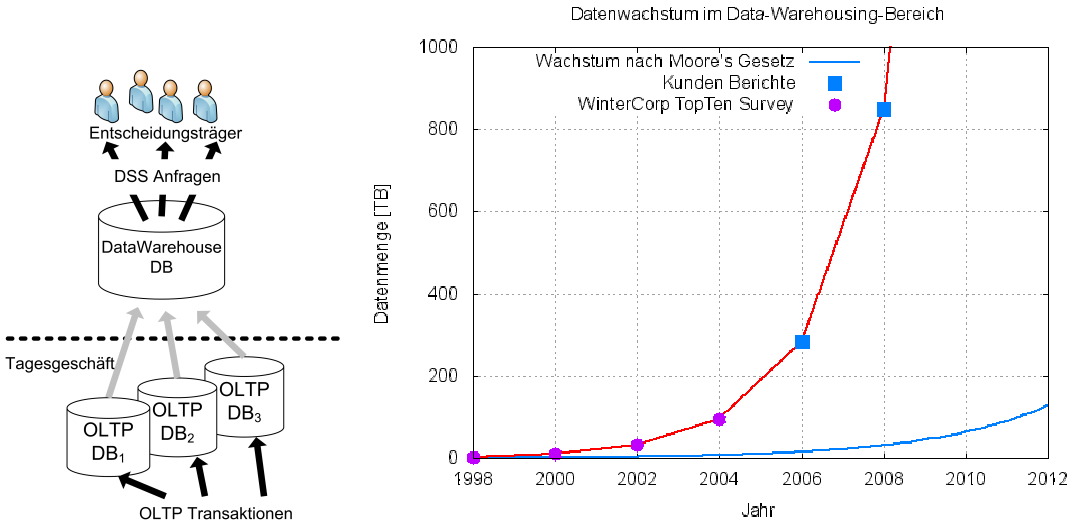
OLTP – OnLine Transaction Processing

Ein Data-Warehouse dient der Datenanalyse (OLAP) und als System zur Entscheidungshilfe für betriebswirtschaftliche Entschlüsse (DSS) [88]. DSSe und Data-Warehouses sind über die letzten Jahre zu mächtigen Werkzeugen für Analysten und Manager geworden. Dabei werden große Datenmengen in komprimierter Form dargestellt und den Benutzern Möglichkeiten gegeben, komplexe Anfragen an das RDBMS zu stellen. Diese Anfragen sind zum einen rechenintensiv, zum anderen werden meist viele Tupel aus verschiedenen Relationen miteinander in Beziehung gesetzt. Im Gegensatz dazu werden bei OLTP-Anfragen meist nur wenige oder sogar nur einzelne Tupel gelesen bzw. geschrieben.

OLAP – OnLine Analytical Processing

DSS – Decision Support System

2005 hat die Winter-Corporation in einer Untersuchung die Größe und den Wachstumsfaktor von Datenmengen in Data-Warehouse-Systemen betrachtet [169]. Demnach verdreifachten sich die Daten etwa alle zwei Jahre in der letzten Dekade und viele Indikatoren weisen auf ein weiteres Wachstum mit der gleichen oder einer höheren Rate in der kommenden Dekade hin [170]. Im April 2009 veröffentlichte Curt Monash [125] einen Artikel über zwei große Data-Warehouses bei eBay. Das erste wird mit einer Grösse von mehr als zwei PByte an Daten, mehrere Zehntausend Nutzer und viele Millionen



(a) Schematische Darstellung (b) Entwicklung der Datenmengen [170]

Abbildung 1.2: Data-Warehouse

Anfragen pro Tag angegeben. Es ist auf 72 Knoten verteilt und wird von mehreren hundert Datenbanken gespeist. Das zweite Data-Warehouse umfasst 6,5 PByte an Daten in etwa 17 Billionen Datensätzen. Es wächst mit einer Rate von 150 Milliarden neuen Datensätzen pro Tag (50 TByte pro Tag) und ist auf 96 Knoten verteilt.

Ein Data-Warehouse ist typischer Weise nach einem **Sternschema** oder dem verwandten **Schneeflocken-Schema** modelliert. Bei einem Sternschema formieren sich verschiedene **Dimensionstabellen** sternartig um die zentrale **Faktentabelle**. Die Faktentabelle nutzt Fremdschlüsselbeziehungen, um auf Daten aus den einzelnen Dimensionstabellen zu referenzieren. Abbildung 1.3 zeigt ein Data-Warehouse-Schema mit der zentralen Faktentabelle *LineItem*; die restlichen dargestellten Tabellen sind die Dimensionstabellen.

1.1.2 Der TPC-H-Benchmark

TPC – Transaction Processing Performance Council

TPC-H – TPC ad Hoc

Das auf Abbildung 1.3 dargestellte Datenbankschema, ist das Schema[1] des **TPC-H-Benchmarks** [162]. Der TPC-H-Benchmark stellt ein fiktives Data-Warehouse dar, mit dem Leistungsvergleiche und Bewertungen von RDBMSen vorgenommen werden können. Es instantiiert Kunden (*Customer*), Lieferanten (*Supplier*), Teile (*Part*) und Bestellungen (*Orders*). Eine Bestellung eines Kunden besteht aus vielen Einzelposten (*LineItem*). Jeder Einzelposten stellt eine Menge eines bestimmten Teils von einem Lieferanten (*PartSupp*) dar. Kunden und Lieferanten sind einer bestimmten Nation (*Nation*) zugehörig; jede Nation liegt in einer bestimmten Region (*Region*).

[1] Den Attributnamen wird noch der jeweilige Tabellenprefix vorangestellt

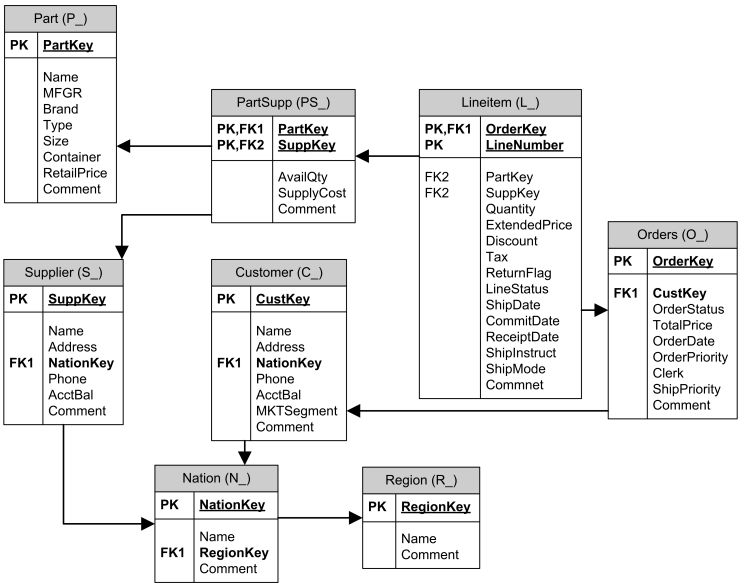


Abbildung 1.3: Schema des TPC-H-Benchmarks [162]

Neben dem Datenbankschema stellt der TPC-H-Benchmark auch ein Datengenerierungswerkzeug und eine Menge von 22 Anfragen bereit. Diese Anfragen sollen dem System nicht im Vorhinein bekannt sein, sondern stellen vielmehr Ad-hoc-Anfragen dar.

Beispiel 1.1: TPC-H-Anfrage Q_9

```
select nation, o_year, sum(amount) as sum_profit
 from ( select n_name as nation,
    extract(year from o_orderdate) as o_year,
    l_extendedprice * (1 - l_discount) -
    ps_supplycost * l_quantity as amount
   from part, supplier, lineitem, partsupp, orders, nation
   where s_suppkey = l_suppkey
    and ps_suppkey = l_suppkey
    and ps_partkey = l_partkey
    and p_partkey = l_partkey
    and o_orderkey = l_orderkey
    and s_nationkey = n_nationkey
    and p_name like '%[COLOR]%' ) as profit
 group by nation, o_year
 order by nation, o_year desc;
```

Beispiel 1.1 zeigt die SQL-Syntax der neunten Anfrage und berechnet den Gewinn der einzelnen Produktlinien aufgeschlüsselt nach Nationen und Jahr. Eine Produktlinie besteht aus allen Teilen, die ein bestimmtes Kriterium – parametrisiert über [COLOR] – an ihrem Namen (name **like** '\%[COLOR]\%') erfüllen.

SQL – Structured Query Language

Um dem Nutzer bei seinen Entscheidungen die bestmögliche Hilfe zu geben, ist eine effiziente Bearbeitung von einzelnen komplexen Anfragen auf großen Datenmengen im OLAP-Bereich unabdingbar.

Definition 1.2 (Effiziente Bearbeitung) Eine **effiziente Bearbeitung** bezieht sich in dieser Arbeit auf eine möglichst geringe Ausführungszeit der einzelnen Anfragen. □

1.1.3 Parallele Hardwarearchitekturen

Parallelität kann verwendet werden, um Anfragen auf große Datenmengen effizient zu bearbeiten (Def. 1.2). Parallele Hardwarearchitekturen stellen eine Klasse, der von Flynn [52, 51] aufgestellten Klassifizierung von Rechnerarchitekturen, dar.

SISD – Single Instruction, Single Data

- Bei **SISD** wird ein einzelner Befehl auf einem einzelnen Datum ausgeführt. Hierbei handelt es sich um den klassischen „von-Neumann-Rechner“.

SIMD – Single Instruction, Multiple Data

- Bei **SIMD** wird ein einzelner Befehl auf mehreren unterschiedlichen Daten ausgeführt. Viele moderne Prozessoren haben spezielle Ausführungseinheiten auf dem Prinzip von SIMD.

MISD – Multiple Instruction, Single Data

- **MISD** steht dafür, dass mehrere Befehle auf dem gleichen Datum ausgeführt werden.

MIMD – Multiple Instruction, Multiple Data

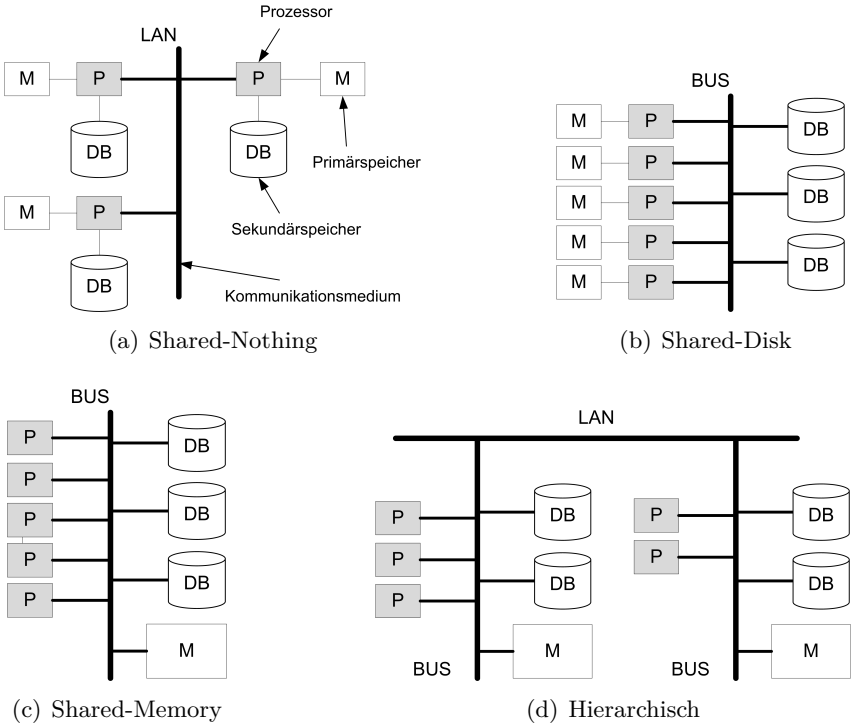
- Die letzte Klasse ist **MIMD**; hierbei werden mehrere Befehle auf mehreren unterschiedlichen Daten ausgeführt. Verteilte Systeme, aktuelle Parallelrechner, Cluster von PCs und auch Mehrkern- und Mehrprozessor-Systeme fallen in diese Kategorie von Rechnerarchitekturen. Bei dieser Kategorie unterscheidet man Architekturen **mit** (**Shared-Memory**) und **ohne gemeinsamen Speicher** (**Distributed-Memory**).

Abbildung 1.4 zeigt eine Klassifikation von parallelen Hardwarearchitekturen. Unterschieden werden [47]: (a) **Shared-Nothing-**, (b) **Shared-Disk-**, (c) **Shared-Memory-** und (d) **hierarchische Architekturen**. Hierarchische Architekturen bilden eine Hybrid-Architektur aus der Shared-Memory- und der Shared-Nothing-Architektur [155].

Hinzu kommen die neuen Mehrkern-Rechnerarchitekturen, die Shared-Memory-Architekturen darstellen. Sich aber bei näherer Betrachtung deutlich von herkömmlichen Shared-Memory-Architekturen unterscheiden. Wichtige Unterschiede sind:

1. Grad der Parallelität
2. Art der Kommunikation
3. Teilung von Ressourcen

Bisherige Shared-Memory-Architekturen bestehen aus einer relativ kleinen Anzahl von Einzelkern-Prozessoren, die über einen oder mehrere Busse miteinander kommunizieren. Die Kommunikation kann abhängig vom Bustyp unterschiedlich Zeit und Ressourcen



(a) Shared-Nothing (b) Shared-Disk

(c) Shared-Memory (d) Hierarchisch

Abbildung 1.4: Parallele Hardwarearchitekturen

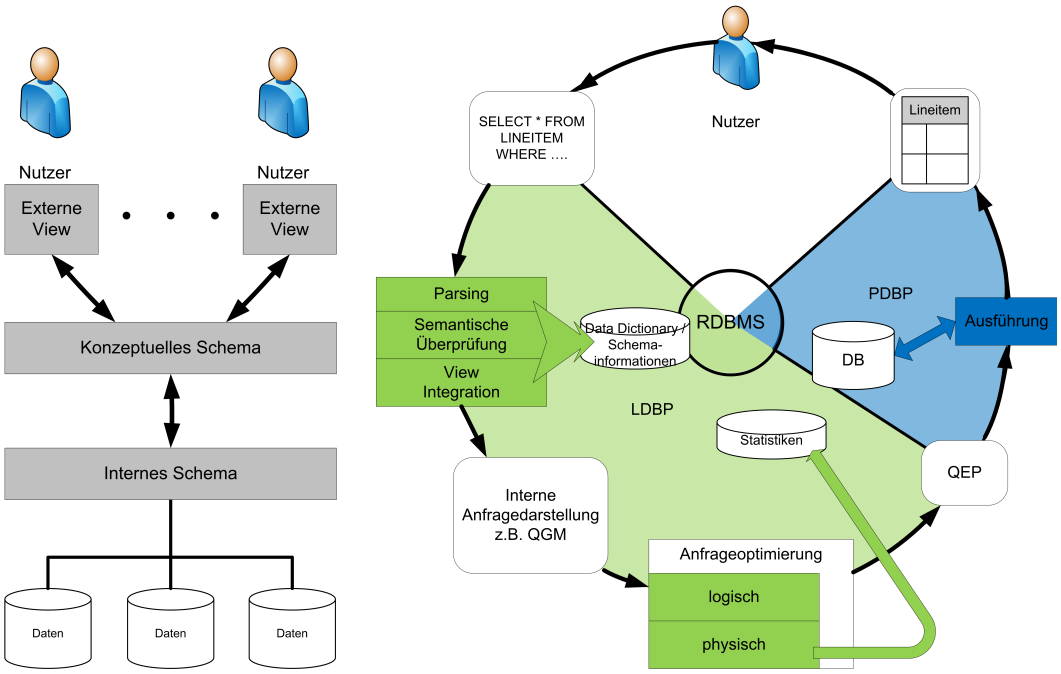
aufwendig sein. Da die einzelnen Recheneinheiten physisch von einander getrennt sind, müssen sie meist nur die Kommunikationsbusse miteinander teilen. Ressourcen des Prozessor (z.B. Caches und Bandbreite) werden nicht geteilt.

Bei Mehrkern-Rechnerarchitekturen dagegen wird eine Vielzahl von Kernen in einem Prozessor untergebracht. Der Grad an Parallelität eines einzelnen Prozessors wird den Grad heutiger paralleler Architekturen übersteigen. Die einzelne Kerne können innerhalb eines Prozessors direkt und mit extrem hoher Geschwindigkeit Daten und Nachrichten austauschen, müssen sich aber alle (bzw. Teilmengen aller) Ressourcen teilen. Die Herausforderungen für zukünftige RDBMSe bestehen in der effizienten Nutzung der vorhandenen Parallelität der Mehrkern-Rechnerarchitekturen. Mit genau diesem Thema wird sich diese Arbeit befassen und mögliche Lösungen und Lösungsansätze präsentieren.

1.2 Anfragebearbeitung in RDBMSen

Im Folgenden werden kurz wichtige Konzepte und grundlegende Begriffe eingeführt, um eine Grundlage und ein besseres Verständnis der Zusammenhänge zwischen Anfrage, Daten, Optimierung und Ausführung zu geben. Eine weit tiefere Erklärung und Beschreibung von Konzepten und Techniken folgt in den jeweiligen Kapiteln 3, 4 und 5.

Eine wichtige Eigenschaft von RDBMSen ist die **Datenabstraktion**. Dem Nutzer



(a) 3-Schichten-Architektur (b) Anfragebearbeitung

Abbildung 1.5: Relationale Datenbankmanagementsysteme

wird durch das RDBMS eine **konzeptuelle Sicht** auf die Daten gegeben, bei der der Nutzer keinerlei Informationen über die Art der Speicherung der Daten und die Implementierung von Operatoren auf den Daten benötigt.

Definition 1.3 (3-Schichten-Architektur) Zur Realisierung einer Datenabstraktion wird in RDBMSen eine **3-Schichten-Architektur** verwendet [49, 103]. Sie besteht aus der **externen Schicht**, der **konzeptuellen Schicht** und der **internen Schicht** (s. Abb. 1.5(a)). □

- Die **externe Schicht** oder **Viewebene** definiert die logische Sicht des Anwendungsprogramms oder des Nutzers auf die Daten. Dabei werden nur die für den Nutzer wichtigen bzw. notwendigen Daten dargestellt.
- Die **konzeptuelle Schicht** beschreibt die logische Gesamtheit aller gespeicherter Daten und ist dabei unabhängig von der internen und der externen Schicht. Die logische Struktur wird von einem von den Speicherstrukturen abstrahierendes (logisches) Datenmodell definiert. Im Falle eines RDBMSs ist es das Relationale-Datenmodell.
- Mit der **internen Schicht** wird beschrieben, wie die Daten physisch gespeichert werden und mit welchen physischen Operatoren darauf gearbeitet werden kann. Die interne Schicht wird über das **physische Datenmodell** definiert.

1.2.1 Der Prozess der Anfragebearbeitung

Definition 1.4 (Anfragebearbeitung) Anfragebearbeitung ist ein definierter Prozess (s. Abb. 1.5(b)), den ein RDBMS durchläuft, um eine vom Nutzer an das RDBMS übergebene Anfrage zu beantworten. □

Der Nutzer definiert seine Anfrage auf Objekten der externen Schicht. Diese Anfrage wird durch das RDBMS während der Anfragebearbeitung erst in eine Anfrage auf der konzeptuellen, dann in Operatoren der internen Schicht transformiert. Anschließend werden die Ergebnisse in der internen Schicht berechnet und dann zurück über die konzeptuelle in die externe Ebene transformiert.

Der Vorgang der Anfragebearbeitung kann in zwei Phasen zerlegt werden, die jeweils in weitere Teilphasen zerfallen [157]. Die Anfrage wird in einer deklarativen Sprache (z.B. SQL, s. Bsp. 1.1) spezifiziert. Dabei enthält die Anfrage keinerlei Information, wie die Anfrage ausgeführt werden soll; es ist vielmehr eine Beschreibung des zu erwartenden Ergebnisses.

In der ersten Phase wird die Anfrage durch den **logischen Datenbankprozessor** (LDBP) in ein prozedurales Programm überführt. Der Ablauf innerhalb des LDBP stellt sich wie folgt dar:

LDBP – Logical DataBase Processor

1. Parsing, d.h. syntaktische Überprüfung
2. Semantische Überprüfung
3. View-Integration
4. Anfrageoptimierung
   a) logische Anfrageoptimierung
   b) physische Anfrageoptimierung

Das **Parsing** und die **semantische Überprüfung** überführen die Anfrage in eine interne Repräsentation. Dabei wird geprüft, ob die Anfrage syntaktisch korrekt ist und ob sie bzgl. der Datenbank eine semantisch sinnvolle Anfrage darstellt. Es folgt die **View Integration**, die gegebenenfalls verwendete Sichten (Views) durch ihre Definitionen ersetzt. Am Ende dieser Verarbeitung steht eine äquivalente interne Repräsentation der Anfrage z.B. in Form von QGM [64].

QGM – Query Graph Model

Die interne Repräsentation wird danach weiter an die **Anfrageoptimierung** gegeben. Sie ist dafür zuständig, einen möglichst optimalen und effizienten (s. Def. 1.2) Anfrageausführungsplan (QEP) zu erzeugen.

QEP – Query Execution Plan

Definition 1.5 (Anfrageausführungsplan) Ein **QEP** ist ein gerichteter, azyklischer Operatorgraph (DAG). Seine internen Knoten sind algebraische Operatoren; die Blätter stellen Zugriffe auf Relationen dar; während die Kanten den Datenfluss von den Blättern zur Wurzel wiedergeben. □

DAG – Directed Acyclic Graph

Die Aufgabe der Anfrageoptimierung ist keineswegs trivial, da für eine gegebene Anfrage eine große Anzahl an potenziellen QEPs existieren kann. Die Phase der Anfrageoptimierung erfolgt ebenfalls in mehreren Stufen: Der (i) **logischen Optimierung**, mit Normalisierung und Vereinfachung; gefolgt von der (ii) **physischen Optimierung**, mit Graphtransformationen und Algorithmenauswahl [54].

PDBP – Physical DataBase Processor

In Phase zwei der Anfragebearbeitung führt der **physische Datenbankprozessor** (PDBP) den erzeugten QEP auf der Datenbank aus und gibt die Ergebnisse zurück (vgl. Abb. 1.5(b)). Für eine effiziente und skalierbare Anfragebearbeitung ist ein enges Zusammenspiel zwischen dem **physischen Datenmodell**, der **Anfrageoptimierung** und **Anfrageausführung** notwendig.

1.2.2 Anfragebearbeitung in parallelen RDBMSen

Das vorrangige Ziel der parallelen Anfrageausführung ist es, Leistungsvorteile durch die Ausnutzung der Parallelität zu erzielen. Dabei werden Leistungsvorteile hauptsächlich über zwei Maßzahlen erfasst. Die Erste, der **Durchsatz**, gibt an, wie viele Anfragen innerhalb einer bestimmten Zeit komplett ausgeführt werden konnten. Die zweite Maßzahl ist die **Antwortzeit**, sie erfasst, wie lange eine gegebene Anfrage braucht, um vollständig ausgeführt zu werden.

OLTP-Systeme haben das Ziel, einen möglichst hohen Wert für den *Durchsatz* zu erreichen. Ein RDBMS, das viele kleine Anfragen verarbeiten muss, kann mit Hilfe der sogenannten **Interquery-Parallelität** den Durchsatz erhöhen.

Definition 1.6 (Interquery-Parallelität) Die parallele Abarbeitung von mehreren verschiedenen Anfragen wird als **Interquery-Parallelität** bezeichnet [158]. □

Im Gegensatz dazu ist in einem OLAP/DSS-System die Antwortzeit die vorrangige Maßzahl. Ein OLAP-System mit komplexen Anfragen kann die Antwortzeit durch Verwendung von **Intraquery-Parallelität** verbessern.

Definition 1.7 (Intraquery-Parallelität) Auf die Nutzung der parallelen Verarbeitung innerhalb einer **einzelnen** Anfrage wird mit **Intraquery-Parallelität** verwiesen [158]. □

Der Durchsatz und die Antwortzeit werden durch die Maßzahlen **SpeedUp** und **Scale** erfasst.

Definition 1.8 (*SpeedUp*) Das Verhältnis zwischen der Ausführungszeit ohne Parallelität und mit Parallelität wird als *SpeedUp* bezeichnet.

$$SpeedUp = \frac{\text{Ausführungszeit eines Prozessors}}{\text{Ausführungszeit bei N Prozessoren}}$$ □

Wächst ein Leistungsgewinn linear mit der Anzahl der Prozessoren, wird dies als **linearer** *SpeedUp* bezeichnet. In seltenen Fällen – z.B. durch Cacheeffekte – kann es zu **superlinearen** *SpeedUp* ($SpeedUp > N$) kommen.

Definition 1.9 (*Scale*) *Scale* spiegelt die Fähigkeit eines Systems wider, sich an wachsende Aufgaben durch wachsende Systeme anpassen zu können. Es wird die Ausführungszeit einer kleinen Aufgabe in einem kleinen System mit der Ausführungszeit einer größeren Aufgabe in einem größeren System miteinander in Beziehung gesetzt.

$$Scale = \frac{\text{Ausführungszeit auf einem kleinen System mit kleiner Aufgabe}}{\text{Ausführungszeit auf einem großen System mit großer Aufgabe}}$$ □

Linearer *Scale* beschreibt ein gleich bleibendes Leistungsverhalten eines Systems, wenn Aufgaben und Ressourcen im gleichen Verhältnis zueinander vergrößert werden. Ein RDBMS kann **vertikal** (*ScaleUp*) oder **horizontal** (*ScaleOut*) wachsen. Beispielsweise ist der Übergang von einem Einzelkern-System auf ein Mehrkern-System eine vertikale Skalierung. Beim *ScaleOut* wird das System horizontal skaliert. Anstatt das bestehende System, wie beim *ScaleUp* durch ein leistungsfähigeres System zu ersetzen, wird es durch zusätzliche Komponenten erweitert.

Im Rahmen von parallelen RDBMSen wird bzgl. der Skalierung der Aufgabe wie folgt unterschieden: (i) **Transaktionsskalierung** und (ii) **Datenskalierung**. Bei der Transaktionsskalierung erhöht sich die Anzahl der Transaktionen pro Zeiteinheit, wogegen sich bei der Datenskalierung die Größe der Datenbank erhöht, so dass mehr Daten pro Anfrage verarbeitet werden müssen.

Linearer *Scale* und *SpeedUp* ist das oberste Ziel der parallelen Anfragebearbeitung in RDBMSen. Eine Menge von Faktoren behindern aber die effiziente, parallele Ausführung und vermindern damit *SpeedUp* und *Scale* [161]. Zu ihnen gehören insbesondere: (i) Start- und Endkosten, (ii) Interferenz und Kommunikation sowie (iii) die Ungleichverteilung in den Daten. Lu und Ooi stellen weitere detaillierte Konzepte und Ansätze der parallelen Anfragebearbeitung in [111] dar.

1.3 Ziele der Arbeit

Mit der vorliegenden Arbeit sollen Möglichkeiten zur effizienten Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen untersucht werden. Es werden datenintensive Anfragen betrachtet, wie sie im OLAP/DSS-Umfeld vorkommen. Dabei wird sich diese Arbeit auf Shared-Memory-Architekturen mit Mehrkern-Prozessoren beschränken.

Ziel der Arbeit ist die Entwicklung neuer Konzepte und Algorithmen bzw. die Anpassung vorhandener Konzepte und Algorithmen im Bereich der Anfragebearbeitung in RDBMSen. Die entwickelten Ansätze sollen miteinander verknüpft und in einem Rahmen (engl. Framework) zusammengefasst werden. In [133] definiert Panchal ein Framework als:

> "'A Framework is an incomplete, though concrete, driving solution to recurring high-value problems."'

Definition 1.10 (Framework) Ein **Framework** ist eine Menge von Konzepten und Techniken. Obwohl es unvollständig ist, ist es dennoch konkret und führt zu Lösungen von sich wiederholenden und wichtigen Problemen. □

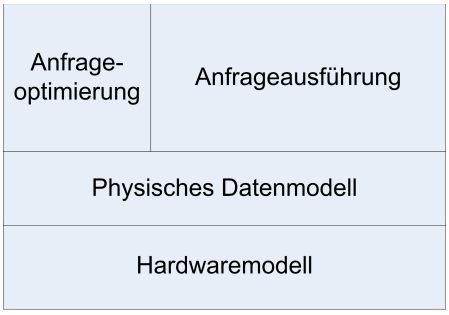


Abbildung 1.6: Framework zur Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen

Abbildung 1.6 zeigt die Struktur des in dieser Arbeit entwickelten Frameworks und seine vier Teilbereiche: ein **Hardwaremodell**, ein **physisches Datenmodell**, die **Anfrageausführung** und die **Anfrageoptimierung**. Innerhalb der einzelnen Teilbereiche werden dazu Konzept und Methoden vorgestellt, die dazu dienen eine effiziente Anfragebearbeitung im Kontext von RDBMSen auf Mehrkern-Rechnerarchitekturen umzusetzen. Das Framework wird als Grundlage für eine prototypische Realisierung verwendet, die als Nachweis für die Funktionsfähigkeit und als Praxistest dient. Die vorliegende Arbeit hat dabei folgende Teilziele:

- Die Entwicklung eines geeigneten Hardwaremodells, das es erlaubt, die vorhandenen Möglichkeiten und Eigenschaften der Hardware optimal zu nutzen.
- Die Formulierung von speziellen Operatoren als Teil des Hardwaremodells.
- Eine Überprüfung und Erweiterung vorhandener physischer Datenmodelle bzgl. ihrer Eignung für die Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen im Data-Warehouse-Bereich.
- Die Entwicklung eines Indexes, der cacheeffizient ist und die vorhandenen Möglichkeiten der heutigen Mehrkern-Prozessoren optimal nutzen kann.
- Die Entwicklung eines neuen Anfrageausführungsmodells für Mehrkern-Rechnerarchitekturen.
- Die Konzeption der Anfrageoptimierung für Mehrkern-Rechnerarchitekturen bzgl. der Ressourcenverwaltung.
- Eine Realisierung der entwickelten Konzepte in einem funktionsfähigen Prototypen.

Bei dem Entwurf und der Entwicklung des Frameworks ist es wichtig, dass die einzelnen Teilbereiche optimal aufeinander abgestimmt sind. Dazu ist es notwendig, den gesamten Prozess der Anfragebearbeitung zu analysieren und zu verstehen.

Das Ziel dieser Arbeit ist dabei nicht, in jedem Bereich neue Algorithmen und Methoden zu entwickeln, sondern viel mehr vorhandenes auf ihre Mehrkern-Tauglichkeit zu prüfen und gegebenenfalls anzupassen oder zu ersetzen.

Die Herausforderungen dieser Arbeit liegen in mehreren Bereichen. Zunächst ist ein detailliertes Wissen über die einzelnen Teilbereiche der Anfragebearbeitung und ihrer Zusammenhänge notwendig. Dazu kommt das Verständnis für Mehrkern-Rechnerarchitekturen und die Probleme, die durch die unterschiedlichen Verwendungsmöglichkeiten entstehen. Zuletzt ist es unabdingbar, die entwickelten Konzepte und Algorithmen mit Hilfe eines passenden parallelen Programmiermodells umzusetzen.

Aus diesem Grund wird sich ein gewisser Teil dieser Arbeit mit der Zusammenfassung und Darstellung des notwendigen Wissen befassen. Dieses gilt speziell für den Teil der sich mit Hardwaredetails und Mehrkern-Rechnerarchitekturen befasst. Damit soll auch dem Leser, der vielleicht nicht dieses detailliertes Wissen besitzt, die Möglichkeit gegeben werden, diese Arbeit ohne weiteres Literaturstudium zu verstehen. Das Verständnis der vorliegenden Arbeit setzt aber ein Grundverständnis für Computer und Datenbanken voraus.

1.4 Struktur der Arbeit

Im Folgenden wird die Struktur der Arbeit vorgestellt. Das Ziel der Arbeit ist mit der Schaffung eines Frameworks zur Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen gegeben, dazu wird von Kapitel zu Kapitel das Framework Schritt um Schritt definiert und erweitert. Mit Ausnahme von Kapitel 1 und Kapitel 6 weisen alle Kapitel eine einheitliche Struktur auf. Zu Beginn eines jeden Kapitels wird das entsprechende Teilgebiet eingeführt und erläutert. Es werden existierende Forschungsarbeiten, bekannte Konzepte sowie Begrifflichkeiten vorgestellt. Leser mit detailliertem Wissen im jeweiligen Gebiet können dabei durchaus Teile überspringen und zum nächsten Teil übergehen. Im letzten Teil eines jeden Kapitels werden die im Rahmen dieser Arbeit entwickelten Konzepte und Umsetzungen ausführlich vorgestellt und beschrieben.

Kapitel 1 – Einleitung

Im ersten Teil des ersten Kapitels wurde der Inhalt der vorliegenden Arbeit motiviert. Dazu wurde das Problem der Anfragebearbeitung im Data-Warehouse-Bereich anhand des TPC-H-Benchmarks vorgestellt. Der zweite Teil gab eine Zusammenfassung über die Anfragebearbeitung in RDBMSen und wurde vom Abschnitt gefolgt, in dem die Ziele und Herausforderungen der Arbeit beschrieben wurden. Im letzten Teil des Kapitels wird die Struktur der vorliegenden Arbeit vorgestellt.

Kapitel 2 – Mehrkern-Rechnerarchitekturen

Im zweiten Kapitel der Arbeit sollen die Grundlagen von Mehrkern-Rechnerarchitekturen erläutert werden. Die Arbeit setzt ein sehr gutes Verständnis bzgl. Aufbau, Funktionsweise und Problemen von Mehrkern-Rechnerarchitekturen voraus und wird sich

daher im ersten Teil sehr ausführlich diesem Thema widmen. Im Vordergrund stehen Prozessoren und Speicher; den Abschluss stellt ein Ausblick auf die Eigenschaften und den Aufbau von zukünftigen Mehrkern-Prozessoren dar.

Der zweite Abschnitt stellt Konzepte der parallelen Programmierung und unterschiedliche parallele Programmiermodelle in den Vordergrund. Unter anderem werden verschiedene Möglichkeiten gezeigt, parallele Programme zu schreiben.

Der letzte Teil des dritten Kapitels beschreibt den ersten Teil des Frameworks. Um die notwendigen Gemeinsamkeiten der unterschiedlichen Mehrkern-Rechnerarchitekturen darzustellen und einen einfachen Blick und Umgang zu ermöglichen, wird ein Hardwaremodell eingeführt. Als Teil von diesem Modell werden eine Reihe von Operatoren erstellt, welche später von unterschiedlichen Teilen des Frameworks genutzt werden, um eine effiziente Anfragebearbeitung zu ermöglichen.

Kapitel 3 – Das physisches Datenmodell

Aufbauend auf dem Hardwaremodell und seinen Operatoren, stellt das physische Datenmodell die zweite Ebene im Framework dar. Aus diesem Grund, wird das Kapitel 3 das physischen Datenmodell einführen und diskutieren.

Im ersten Teil wird das Relationenmodell und die relationale Algebra vorgestellt, dem eine Diskussion über verschiedene Umsetzungen des physischen Datenmodells im zweiten Abschnitt folgt. Teil 3 des dritten Kapitels präsentiert das physisches Datenmodell des zu entwickelnden Framework. Dazu wird das Konzept der vertikalen Partitionierung erweitert. Abschließend werden für die Anfrageausführung notwendige Operationen auf dem Datenmodell definiert und beschrieben.

Im vierten Teil wird zunächst ein Indexierungsverfahren vorgestellt, das die Möglichkeiten der Vektorverarbeitung (SIMD) heutiger Prozessoren besser nutzen kann. Ausgehend von Messungen wird dann ein neues im Rahmen dieser Arbeit entstandenes Indexierungsverfahren präsentiert, das hohe Leistungsgewinne durch die Verwendung von SIMD-Operationen verspricht.

Kapitel 4 – Anfrageausführung

Kapitel 4 behandelt das Thema der Anfrageausführung. Dazu werden zunächst Grundlagen der Anfrageausführung dargestellt, der eine Beschreibung und ein Vergleich existierender Anfrageausführungsmodelle in Abschnitt 2 folgt. Die Vor- und Nachteile der einzelnen Modelle dienen als Ausgangspunkt für den dritten Abschnitt.

Im dritten Abschnitt wird, als Teil des zu entwickelnden Frameworks, ein neues Anfrageausführungsmodell beschrieben, welches auf dem Konzept von Tasks beruht und einen hohen Grad an Intraquery-Parallelität (Def. 1.7) ermöglicht. Die Ausführung verwendet dabei die Operatoren des physischen Datenmodells und das Hardwaremodell für die Ressourcenverwaltung und -zuordnung.

Kapitel 5 – Anfrageoptimierung

Der Prozess der Anfrageoptimierung steht im fünften Kapitel im Vordergrund. Im ersten Teil werden zunächst bekannte Konzepte der Anfrageoptimierung für die sequentielle und parallele Anfragebearbeitung vorgestellt, gefolgt von Konzepten der adaptiven Anfragebearbeitung im zweiten Abschnitt. Zum Abschluss wird der letzte Teil des Frameworks vorgestellt, darin werden verschiedene Ansätze zur Anfrageoptimierung im Kontext der Ressourcenverwaltung präsentiert.

Kapitel 6 – Zusammenfassung und Ausblick

Das siebente und letzte Kapitel fasst die Ergebnisse der vorliegenden Arbeit zusammen. Anhand der im ersten Kapitel definierten Ziele wird die Arbeit kritisch evaluiert und diskutiert. Abschließend werden noch offene Punkte sowie weitergehende Forschungsfragen vorgestellt.

Am Ende der Arbeit befinden sich eine Übersicht über Akronyme und Abkürzungen, sowie die Verzeichnisse zur verwendeten Literatur, Definitionen, Abbildungen, Tabellen und Beispielen.

2 Mehrkern-Rechnerarchitekturen

"'There are only two kinds of languages: the ones people complain about and the ones nobody uses."'

Bjarne Stroustrup (1950–)

Kapitel 2 stellt aktuelle und zukünftige Rechnerarchitekturen vor. Das Hauptaugenmerk liegt dabei auf Hauptspeicher, Prozessoren und deren Caches. Dieser Teil ist zum Verständnis von Problemen, möglichen Lösungen und zur Interpretation von Ergebnissen der Arbeit maßgeblich und wird daher sehr detailliert ausfallen.

Um im zweiten Teil dieses Kapitels Konzepte der parallelen Programmierung vorzustellen, werden zunächst wichtige Begriffe wie Prozess, Thread und Task definiert und miteinander in Beziehung gesetzt. Anschließend werden verschiedene Methoden vorgestellt, parallele Programme zu spezifizieren; aus diesem Grund werden verschiedene Erweiterungen, wie z.B. OpenMP und Intel-TBB, für die Programmiersprachen C und C++ präsentiert. Abschließend werden Vor- und Nachteile der unterschiedlichen Erweiterungen erläutert.

Den Abschluss des Kapitels stellt die hardware-naheste Schicht des entwickelten Frameworks (s. Abschn. 1.3) vor. Dazu wird zuerst ein Hardwaremodell und im Anschluss Operatoren definiert, die es ermöglichen die komplexen und heterogenen Eigenschaften von Mehrkern-Rechnerarchitekturen in einer einfachen Art und Weise darzustellen und zu nutzen.

2.1 Hardware

Computer setzen sich aus fünf Basiskomponenten zusammen [135]. Diese Komponenten sind: **Eingabe**, **Ausgabe**, **Speicher**, **Datenpfad** und **Kontrolle**. Der Datenpfad und die Kontrolle werden im Allgemeinen zusammengefasst und als **Prozessor** bezeichnet.

Abbildung 2.1 zeigt das Zusammenspiel der fünf Komponenten. Die **Eingabe** dient als Mechanismus, um Informationen an den Computer zu senden. Dazu werden zumeist Tastatur und Maus verwendet. Die **Ausgabe** ist ein Mechanismus, um Informationen vom Computer an den Nutzer zu senden; z.B. über den Bildschirm. Im **Speicher** befinden sich sowohl die Programme bzw. Teile von Programmen, die gerade ausgeführt werden, als auch die Daten, auf die sie zugreifen. Der **Prozessor**, auch CPU genannt, ist einerseits für den Datenpfad zuständig, also die Ausführung von arithmetischen Ope-

CPU – Central Processing Unit

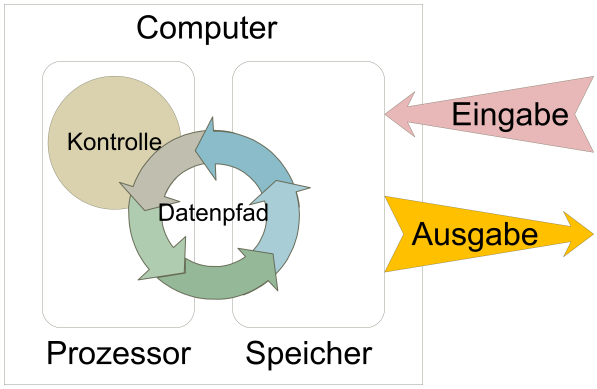


Abbildung 2.1: Fünf Komponenten eines Computers

rationen aber auch für die Kontrolle der Ausführung der Instruktionen des Programms über dem Datenpfad, dem Speicher und der Ein- und Ausgabe. Im Folgenden werden die Teile Speicher und Prozessor ausführlicher betrachtet.

2.1.1 Der Speicher

Der **Speicher** ist der Ort, in dem Programme und ihre Daten liegen und wird in **flüchtigen** und **nicht flüchtigen** Speicher unterschieden. Ein flüchtiger Speicher behält seine Daten nur solange, wie der Computer mit Strom versorgt wird. Dagegen behält ein nicht flüchtiger Speicher seine Daten auch über den Zeitraum ohne Stromzufuhr hinaus. Um auf die unterschiedlichen Speicherarten zu verweisen, werden die Begriffe **Primärspeicher** und **Sekundärspeicher** verwendet. Der Primärspeicher[1] bezeichnet den flüchtigen Teil und der Sekundärspeicher den nicht flüchtigen Teil.

DRAM – Direct Random Access Memory

Typische Vertreter für nicht flüchtigen Speicher sind Magnetbänder und optische Datenträger, wie DVDs sowie Festplatten, darunter magnetische sowie Solid-State-Festplatten. Als flüchtiger Speicher wird in heutigen Computern DRAM eingesetzt. Im Unterschied zu magnetischen Festplatten, Bändern oder optischen Datenträgern, ist der Zugriff beim DRAM auf alle Teile des Speichers gleich schnell.

2.1.1.1 Speicherhierarchien

Unterschiedliche Arten von Speicher haben unterschiedliche Eigenschaften (z.B. Preis pro Megabyte (MB), Zugriffszeit). Aufgrund dieser Eigenschaften entsteht eine sogenannte **Speicherhierarchie** [79, 33, 143], darin bilden preiswerte, große Speichermedien die Basis, nach oben werden die Speicher kleiner und schneller. Am Schluss befinden sich verschiedene Stufen von sogenannten **Caches (Cachehierarchien)** und die internen Prozessorregister.

Definition 2.1 (Cache) Ein **Cache** implementiert das Konzept eines Puffers auf Hardwareebene. Er ist ein kleiner sehr schneller Speicher, der physisch nah am Prozessor bzw.

[1] auch **Hauptspeicher** genannt

im Prozessor liegt (s. Abb. 2.2). Ein Cache bildet keinen zusätzlichen Speicher, sondern bietet Platz für temporäre Kopien von Daten aus dem Primärspeicher. □

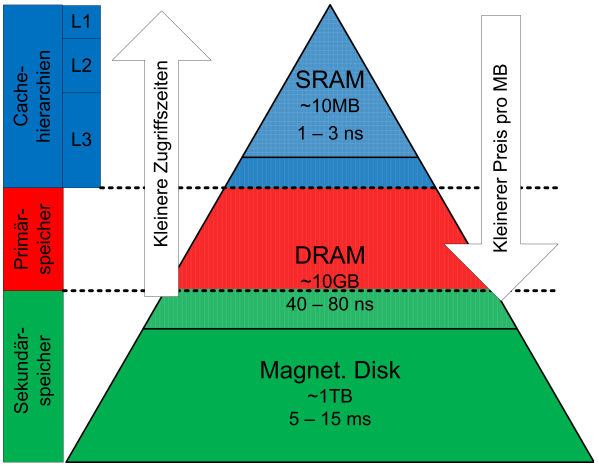


Abbildung 2.2: Darstellung der Speicherhierarchie

Abbildung 2.2 illustriert, die Speicherhierarchie und die jeweiligen Eigenschaften für einen Desktop-Computer im Jahr 2009. Die unterste Ebene bildet der Sekundärspeicher; beispielsweise in Form magnetischer Festplatten mit Größen im Terabyte-Bereich und Zugriffszeiten im Bereich von 5 bis 15 Millisekunden. Preislich betrachtet waren im Jahr 2007 Festplattenspeicher per MB etwa einhundert Mal preiswerter als Speicher auf DRAM-Technologie [135]. Den Bereich darüber bildet der Primärspeicher mit gängige Größen von etwa 10 GByte und Zugriffszeiten im Bereich zwischen 40 und 80 Nanosekunden. Den Abschluss bilden Cachehierarchien, sie werden aus DRAM- oder SRAM-Zellen hergestellt. Eine SRAM-Speicherzelle besteht dabei aus etwa sechs Transistoren und hat Zugriffszeiten von wenigen Nanosekunden. Eine DRAM-Speicherzelle dagegen, besteht nur aus einem einzigen Transistor und ist somit deutlich platzsparender und preiswerter [75]; im Gegenzug haben sie allerdings eine leicht höhere Zugriffszeit (etwa 40 bis 80 Nanosekunden). Die Einführung und Verwendung von Cachehierarchien kann wie folgt begründet werden [33]:

- Es ist preiswerter und technisch einfacher, schnellen Speicher zu konstruieren, wenn er klein ist. Für ein gegebenes Budget kann man entweder eine kleine Menge an sehr schnellem oder eine große Menge an langsamen Speicher erwerben. Eine Kombination ist nicht sinnvoll (s. Bsp. 2.1).

- Ein weiterer Grund ist das Konzept der **Lokalität**[2]. Es existieren **zeitliche** Lokalität sowie **örtliche** Lokalität. Zeitliche Lokalität beschreibt die Eigenschaft, dass ein wiederholter Zugriff auf eine Speicherstelle X innerhalb kurzer Zeit viel wahrscheinlicher ist als auf andere Speicherstellen. Örtliche Lokalität weist auf die

[2]Lokalität ist eine empirische Regel, keine Gesetzmäßigkeit

Eigenschaft hin, dass nach einem Zugriff auf X die Wahrscheinlichkeit sehr hoch ist, dass sehr kurze Zeit später auf eine Speicherstelle nahe X zugegriffen wird.

Beispiel 2.1: Verwendung von Caches

Es soll ein Speicher mit 10-mal mehr langsamen als schnellen Speicher gebaut werden, dabei soll der schnelle Speicher im Vergleich zum langsamen Speicher 10-mal teurer sein. Unter der Annahme, dass die Zugriffe auf den Speicher gleich verteilt sind, folgt, dass 90% aller Zugriffe auf den langsameren Speicher erfolgen. Es wird somit ein maximaler Leistungsgewinn von 10% erzeugt; im Gegenzug sind die Kosten um den Faktor $1,9$ gestiegen. Es ist somit sinnvoller den schnellen Speicher als Cache zu verwenden und das Konzept der *Lokalität* auszunutzen, um damit einen höheren Leistungsgewinn zu erzielen.

Definition 2.2 (Cachezeile) Beim Lesen und Schreiben des Prozessors aus und in den Primärspeicher werden immer Blöcke fester Größe transferiert. Diese Blöcke werden **Cachezeilen** genannt und haben üblicherweise eine Größe zwischen 16 und 256 Byte. □

Mit Hilfe von Cachezeilen wird versucht, über die Eigenschaft der örtlichen Lokalität Leistungsgewinne zu erzielen. Im Rest der Arbeit werden die Eigenschaften eines Caches C wie folgt notiert: (i) $|C|$ gibt die Gesamtgröße des Caches an, (ii) Wird auf Cachezeilengröße mit $\#C$ und (iii) die Gesamtanzahl der Cachezeilen mit $|\#C|$ referenziert, wobei maximal $|\#C| = \frac{|C|}{\#C}$ Cachezeilen gleichzeitig hält.

Definition 2.3 (Cachetreffer) Fragt ein Prozessor eine Speicherreferenz an, wird geprüft, ob die entsprechende Cachezeile bereits im Cache vorhanden ist. Ist sie im Cache vorrätig, kann der Prozessor mit den Daten weiterarbeiten; der Zugriff wird als **Cachetreffer** (engl. Hit) bezeichnet. Anderenfalls, wird bei einem **Fehlschlag** (engl. Miss) eine Kopie der Daten in den Cache übertragen. □

Definition 2.4 (Hit-Miss-Verhältnis) Der Quotient von Cachetreffern zu Cachefehlschlägen beim Laden von Daten durch den Prozessor wird als **Hit-Miss-Verhältnis** bezeichnet. Das Hit-Miss-Verhältnis gibt Auskunft über die Effektivität des Caches; weiterhin ist es ein Maß für die Lokalität der Zugriffe eines Programms auf den Speicher. □

Beim Laden von Daten aus dem Primärspeicher muss entschieden werden, an welche Position die Daten im Cache untergebracht werden sollen, also welche Cachezeile genutzt wird. Die Position im Cache C lässt sich konzeptuell über die Funktion $mapCacheLine$ darstellen.

$$mapCacheLine : X \rightarrow 0, ..., |\#C| - 1$$

Die Funktion $mapCacheLine$ ordnet jeder Speicheradresse X eine aus mehreren möglichen Cachezeilen zu und wird von der Art des verwendeten Caches definiert. Es werden drei Arten unterschieden:

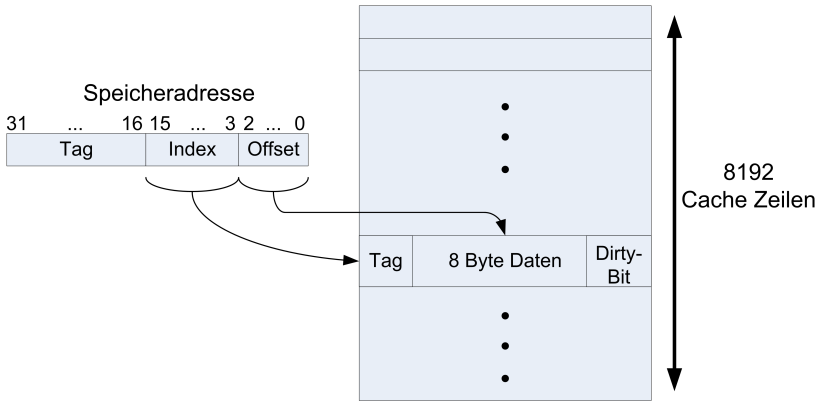


Abbildung 2.3: Direkt adressierter Cache

- Bei einem **direkt adressierten** Cache wird jede Speicherreferenz auf genau eine Cachezeile abgebildet. Bei der Verdrängung besteht somit nicht die Möglichkeit eine andere Zeile auszuwählen. Abbildung 2.3 zeigt eine mögliche Implementierung eines direkt adressierten Caches. Die *mapCacheLine* Funktion benutzt die Bits 15 bis 3 (Index), um die Position der Cachezeile zu berechnen und die Bits 2 bis 0 für die Positionierung (Offset) innerhalb der Cachezeile. Die Bits 31 bis 16 (Tag) werden als Zusatzinformationen an die Cachezeile angehangen, um zu prüfen, ob ein Miss oder ein Hit vorliegt.

- Ein Cache heißt **K-Wege assoziativ**, wenn jede Speicherreferenz auf K der $|\#C|$ Cachezeilen abgebildet werden kann. Welche der K Cachezeilen es letzten Endes ist, wird mit Hilfe einer Verdrängungsstrategie bestimmt, meist wird dabei ein LRU-Algorithmus eingesetzt. Ein K-Wege assoziativer Cache hat durch die Auswahlmöglichkeit ein besseres Hit-Miss-Verhältnis als ein direkt adressierter Cache. Allerdings wächst der Aufwand für das Prüfen, ob es sich um einen Treffer oder einen Fehlschlag handelt linear mit der Anzahl der möglichen Cachezeilen.

  LRU – Least Recently Used

- **Vollständig assoziative** Caches sind K-Wege assoziative Caches mit $K = |\#C|$.

So lange Daten im Cache nur gelesen und nicht verändert werden, müssen bei der Verdrängung keine weiteren Maßnahmen getroffen werden; die Cachezeile wird einfach mit den neuen Daten überschrieben. Werden Daten verändert, müssen diese Veränderungen früher oder später auch im Primärspeicher durchgeführt werden. Dazu existieren zwei Verfahren:

- Beim **direkten Schreiben** werden die Daten synchron in den Cache und in den Primärspeicher geschrieben. Dies hat wiederum zur Folge, dass der Cache und der Primärspeicher immer konsistent sind. Das System kann während des Schreibens weiterarbeiten, allerdings kommt es zu mehr Last auf den Verbindungen vom Primärspeicher zum Cache, da für jede Änderungsoperation die komplette Cachezeile zurückgeschrieben werden muss.

- Beim **verzögerten Schreiben** werden die Daten nicht direkt in den Hauptspeicher geschrieben, wodurch es zu unterschiedlichen Zuständen zwischen Hauptspeicher und Cache kommt (Inkonsistenz). Das sogenannte **Dirty-Bit** (s. Abb. 2.3) zeigt an, ob eine Speicherreferenz verändert wurde oder nicht. Bei der Verdrängung wird anhand des Dirty-Bits ein Zurückschreiben veranlasst. Der Vorteil besteht darin, dass der Prozessor die gleiche Cachezeile mehrmals schreiben kann ohne zusätzlichen Aufwand durch das wiederholte Schreiben in den Primärspeicher zu generieren.

Wenn mehrere Caches als Stufen übereinander angeordnet werden, unterscheidet man **inklusive** und **exklusive Cachehierarchien**. Zwei Cachestufen werden als inklusive bezeichnet, falls alle Daten der oberen Hierarchieebene auch in der unteren vorhanden sind. Daraus folgt, dass bei der Verdrängung einer Cachezeile in der unteren Hierarchie, auch alle Kopien dieser Zeile in den oberen Ebenen als ungültig erklärt werden müssen. Dagegen kann es bei zwei Cachestufen, die exklusiv zu einander stehen, vorkommen, dass in der höheren Stufe eine Cachezeile vorhanden ist, die nicht in der unteren Stufe vorliegt. Das Prüfen auf einen Treffer in Cachehierarchien erfolgt von der obersten Stufe hinunter zur untersten Stufe. Bei einem Fehlschlag auf der letzten Stufe werden dann die Daten aus dem Primärspeicher geladen.

2.1.1.2 Cache-Synchronisation

Bei Computern mit mehreren Prozessoren oder Verarbeitungseinheiten haben die einzelnen Prozessoren oft private Caches oder Cachehierarchien. Aus diesem Grund kann es ohne weitere Vorkehrungen zu Problemen in der Datenverarbeitung durch Inkonsistenzen kommen.

Inkonsistenzen von Daten können entstehen, wenn ein Prozessor in einem Cache mit verzögertem Schreiben Daten verändert, während ein anderer Prozessor diese Daten über einen anderen Cache weiterverarbeiten will. Ohne weitere Vorkehrungen würde der zweite Prozessor mit veralteten, nicht konsistenten Daten arbeiten. Um dieses Problem zu vermeiden, existieren **Cache-Kohärenz-Protokolle**. Diese Protokolle finden Synchronisationskonflikte und stellen für die verschiedenen Prozessoren eine gemeinsame Sicht auf den Speicher her.

Für Cache-Kohärenz-Protokolle existieren zwei Methoden [33]: (i) **Veränderungsbasiert**; dabei wird nach jeder Veränderung der neue Wert an alle anderen Caches gesendet. (ii) **Invalidierungsbasiert**; bei diesem Vorgehen muss ein Prozessor vor der Veränderung eines Datums ein exklusives Zugriffsrecht auf der entsprechenden Cachezeile besitzen. Dieses exklusive Zugriffsrecht kann der Prozessor anfragen, woraufhin alle anderen Caches – außerhalb seiner Cache-Hierarchie – die Cachezeile **löschen** oder **invalidieren**. Neben dem exklusiven Zugriffsrecht existiert noch ein weiteres Recht, bei dem sich die Prozessoren den Zugriff teilen (shared). Dabei kann eine Cachezeile gleichzeitig in mehreren Caches zur lesenden Verarbeitung vorliegen. Eine mögliche Realisierung des invalidierungsbasierten Protokolls ist das MESI-Protokoll [79]. Wenn mehrere Prozessoren gleichzeitig auf den gleichen Daten arbeiten [26], können Cache-Kohärenz-Protokolle

zu signifikanten Leistungsverlusten führen; eine spezielle Form stellt das False-Sharing dar.

Definition 2.5 (False-Sharing) Bei **False-Sharing** bearbeiten zwei oder mehr Prozessoren unterschiedliche **unabhängige** Daten, die fälschlicher Weise innerhalb der gleichen Cachezeile liegen. □

Wenn zwei oder mehr Prozessoren gleichzeitig auf den Daten einer Cachezeile arbeiten, kommt es z.B. beim invalidierungsbasierten Cache-Kohärenz-Protokoll dazu, dass alle Prozessoren um das exklusive Zugriffsrecht auf eine Cachezeile konkurrieren und sich somit blockieren [26, 92]. Neben der Blockierung entsteht auch durch die ständige Invalidierung eine zusätzliche Last bzgl. des Speichertransfers.

Auswirkungen des False-Sharings

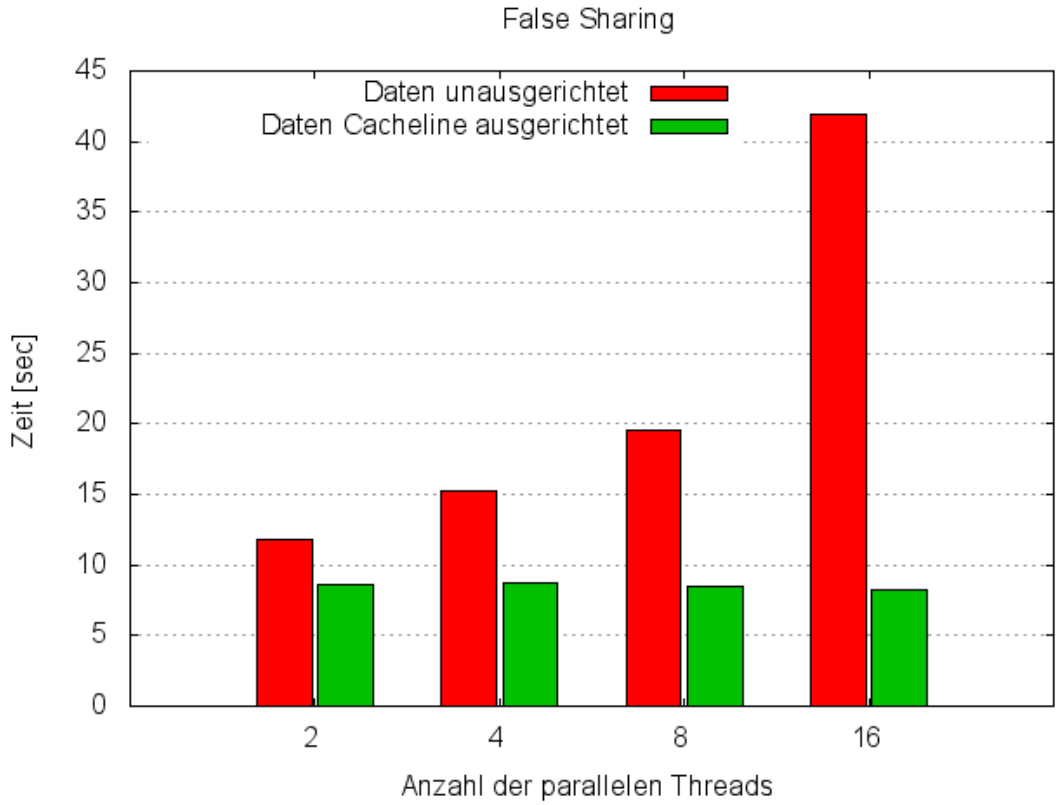


Abbildung 2.4: Leistungsverluste durch False-Sharing

Um die Auswirkungen des False-Sharings quantitativ zu belegen, wurde im Rahmen dieser Arbeit eine Evaluierung auf einem vorliegenden Testsystem (s. Abschn. 2.1.5.4) durchgeführt. Dabei wurden zum einen jeweils zwei, vier, acht oder 16 unabhängige

Integerwerte auf einer Cachezeile angeordnet, zum anderen wurden die Werte so platziert, dass sich keine zwei Werte auf derselben Cachezeile befanden. Anschließend wurde eine entsprechende Anzahl von Threads (Def. 2.16) gestartet; jeder Thread inkrementiert jeweils einen Wert innerhalb einer Schleife. Die Threads wurden auf die einzelnen Hardware-Threads (Def. 2.15) verteilt und entsprechend gebunden. Im Fall von zwei Threads liefen beide Threads auf dem gleichen Kern (Def. 2.10, gleicher L1/L2 Cache), bei vier Threads sind es zwei Kerne des gleichen Prozessors, bei acht Threads alle Kerne bzw. Hardware-Threads eines Prozessors und bei 16 werden alle Hardware-Threads des gesamten Systems verwendet. Im Falle von 16 Threads war somit die Cache-Kohärenz über einen Prozessor hinaus notwendig.

Abbildung 2.4 zeigt die Auswirkungen auf das Leistungsverhalten in Abhängigkeit von der Anzahl von Threads: mit zunehmender Anzahl von Threads nimmt die Laufzeit für den Fall des *False-Sharing (Rot)* deutlich zu. Dagegen bleibt die Laufzeit im Fall, dass kein False-Sharing vorhanden (Grün) ist, annähernd konstant. Wichtig dabei ist die Beobachtung des überlinearen Anstiegs bei der Synchronisation über zwei unterschiedliche Prozessoren hinweg.

2.1.1.3 Das Speichersystem

Das Speichersystem (engl. Memory Subsystem) verbindet den Primärspeicher mit dem Prozessor. Eine wichtige Maßzahl des Speichersystem ist seine maximale Auslastung, für ihre Bestimmung ist (i) seine Bandbreite B und (ii) die Latenz L des Speichers wichtig. Batten et al. definieren aus ihrem Produkt das **Bandwidth-Delay-Produkt** [22]. Um das Speichersystem voll auszulasten, sind demnach mindestens $(B/b) \times L$ unabhängige b Byte große simultane Anfragen notwendig.

Diese Annahme gilt nur, wenn man diese Datenzugriffe am Cache vorbei führen würde. Da in der Folge eines Fehlschlags die komplette Cachezeile gelesen werden muss, wird in dieser Arbeit die maximale Anzahl simultaner Anfragen wie folgt definiert:

SR – Simultaneously Requests

Definition 2.6 (Simultane Anfragen) Die maximale Anzahl von **simultanen Anfragen** (SR) an ein Speichersystem mit der Bandbreite B, der Latenz L und der Cachezeilengröße $\#C$ berechnet sich mit:

$$SR = \frac{B \times L}{\#C}$$

□

Beispiel 2.2: Maximale Auslastung

Ein Speichersystem, mit einer Bandbreite von 3 GByte pro Sekunde und bei einer Latenz von 250 Nanosekunden, benötigt nach Batten et al. [22] bei einer Elementgröße von 4 Byte etwa 200 simultane Anfragen für eine maximale Auslastung. Unter Berücksichtigung der Definition 2.6 und einer Cachezeilengröße von 128 Byte sind aber maximal sechs simultane Anfragen möglich.

Fazit: Caches bieten eine Möglichkeit die Leistungsfähigkeit von Rechnern zu erhöhen, indem sie die Eigenschaften des Konzepts der Lokalität ausnutzen. Bei der Programmierung ist es notwendig, die Daten möglichst physisch dicht beieinander zu speichern. Dabei ist bei der Verarbeitung von Daten durch mehrere Prozessoren zum einen auf das Problem des False-Sharings zu achten; zum anderen darf der negative Einfluss der Cache-Kohärenz nicht vernachlässigt werden.

2.1.2 Der Prozessor

Der Prozessor ist die zentrale Einheit in heutigen Rechnern und aus mehreren Millionen von Transistoren aufgebaut. Die fundamentale Aufgabe eines Prozessors ist die Ausführung von Programmen, welche Sequenzen von im Primärspeicher gehaltenen Instruktionen oder Befehlen darstellen. Die Ausführung einer Instruktion kann in die folgenden vier Teilschritte zerlegt werden:

1. Laden (IF)
2. Dekodieren (ID)
3. Ausführen (EX)
4. Zurückschreiben (WB)

IF – Instruction Fetch
ID – Instruction Decode
EX – Instruction Execute
WB – Write Back

Der erste Schritt ist das **Laden** einer Instruktion durch den Instruktionslader (vgl. Abb. 2.5). Dabei wird eine Instruktion des Programms aus dem Speicher gelesen. Die Lokation der Instruktion gibt dabei der sogenannte **Programmzähler** (PC) an. Der PC zeigt immer auf den Befehl, der als nächster abgearbeitet werden muss. Nach dem Laden des Befehls wird der PC um die Länge des Befehls erhöht. Im Anschluss an das Laden folgt das **Dekodieren** der Instruktion durch den Instruktionsdekodierer. Hierbei wird die geladene Instruktion in Teilanweisungen zerlegt, die dann die **Ausführung** der eigentlichen Operation steuern. Dazu werden verschiedene Teile des Prozessors genutzt. Als Ein- und Ausgabe dienen primär die internen Prozessorregister. Die meisten Operationen werden über die **arithmetische logische Einheit** (ALU) ausgeführt. Sie führt elementare arithmetische und logische Operationen, wie z.B. Addition, Subtraktion, Multiplikation, Division und Vergleiche, durch.

PC – Program Counter

ALU – Arithmetic Logical Unit

Es folgt der Abschluss der Operation, das **Zurückschreiben**. Darin werden die Ergebnisse der Ausführung in den Speicher zurückgeschrieben; dabei kann es sich sowohl um die Register des Prozessors, als auch den Primärspeicher bzw. die Caches handeln. Im Fall, dass die Instruktion Einfluss aus den Datenpfad nimmt, wie z.B. durch Sprünge oder Aufrufe von Funktionen, wird beim Zurückschreiben der PC entsprechend verändert. Nach der Beendigung des vierten Schrittes fängt der Prozess wieder von vorn an, indem die nächste Instruktion im Datenpfad abgearbeitet wird. Für eine detailliertere Beschreibung wird auf Kapitel 5 in [135] verwiesen.

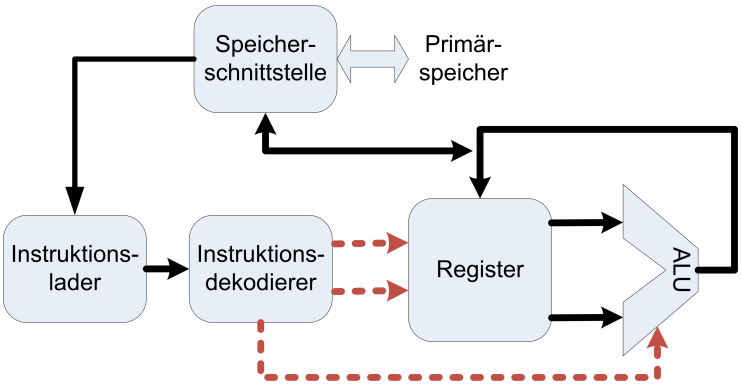


Abbildung 2.5: CPU-Blockdiagramm

2.1.2.1 Kompatibilität von Prozessoren

ISA – Instruction Set Architecture

Programme sollen auf verschiedenen Prozessoren laufen können, ohne sie für jeden einzelnen Prozessor neu übersetzen zu müssen. Dieses Ziel wird damit erreicht, dass unter anderem eine Menge von Instruktionen bzw. Bytefolgen und ihre Bedeutung definiert werden. Diese **definierte Menge von Instruktionen** (ISA) stellt ein Teil der Rechnerarchitektur dar [93] und definiert außerdem native Datentypen, Instruktionen, Register, das Adressierungsmodell und die Speicherarchitektur. Die ISA enthält Spezifikationen über die Maschinenbefehle, auch Opcodes genannt. Eine ISA wird oft über die Zeit um weitere Instruktionen erweitert, bleibt dabei aber kompatibel zu bereits existierenden Programmen. Intel definiert die folgenden drei ISAs[93]:

- Die **IA-32-ISA** ist Grundlage für alle 32 Bit Prozessoren auf Basis von Intel bzw. Intel kompatiblen Prozessoren.
- Eine vollständig neue ISA für 64 Bit Prozessoren stellt die **IA-64-ISA** dar. Sie wird durch die Intel-Itanium-Prozessoren implementiert.
- Eine Erweiterung der IA-32-ISA für 64 Bit Prozessoren ist die **Intel64-ISA**. Damit ist es möglich auf Prozessoren, die die Intel64-ISA implementieren, Programme auszuführen, die für die IA-32-ISA übersetzt wurden.

Die Implementierung einer ISA wird **Mikroarchitektur** genannt und beschreibt das Design, das Layout und die konkrete Implementierung im Silizium [93]. Der Prozessor wird in einem **Rohchip** (engl. Die) implementiert, welcher mit anderen Komponenten im Prozessorgehäuse (engl. Package) untergebracht wird.

Bei der Konstruktion von Prozessoren verfolgt Intel eine Strategie, die mit **Tick-Tock** bezeichnet wird [85, 151]. Das **Tick** bezeichnet die Migration einer bestehenden Mikroarchitektur von einer Prozessgröße auf die nächst Kleinere. Das **Tock** steht für eine Neuentwicklung einer Mikroarchitektur. Zwischen einem Tick und einem Tock liegt in etwa ein Jahr. Mit diesem Zeitrahmen wird etwa alle zwei Jahre eine neue Mikroarchitektur

entwickelt, die im darauf folgenden Jahr auf die nächst kleinere Prozessgröße migriert wird; die Prozessorentwicklung folgt damit dem Mooreschen Gesetz (s. Abschn. 2.1.4).

Die verschiedenen ISAs lassen sich bzgl. der Komplexität der Instruktionen in zwei Gruppen einteilen. Beinhaltet die ISA nur wenige sehr einfache Instruktionen, wird von einer RISC-Architektur gesprochen. Beispiele für moderne RISC-Architekturen sind die Intel-Itanium-Familie, die SUN-Sparc- und die IBM-PowerPC-Prozessoren [142, 79]. Bei der Programmierung von RISC-Prozessoren müssen die komplexen Anweisungen eines Programms in ihre Grundbestandteile zerlegt werden, die ein Äquivalent in der RISC-ISA haben. Im Gegensatz zu RISC-Architekturen haben CISC-Architekturen eine große Menge an komplexen Instruktionen in ihrer ISA. Ihr Vorteil ist die Möglichkeit der direkten Überführung vieler Anweisungen in äquivalente Instruktionen bei der Übersetzung von Programmen.

RISC – Reduced Instruction Set Computer

CISC – Complex Instruction Set Computer

Fazit: Die Tick-Tock-Strategie und die damit verbundene stetige Erweiterung von ISAs innerhalb von kleinen Zeiträumen und die vielen unterschiedlichen Mikroarchitekturen bedeuten eine stetige Anpassung bzw. Überprüfung eines Programms für eine effiziente Programmausführung. Dabei muss nicht nur Detailwissen zum Programm selbst, also bzgl. der verwendeten Algorithmen, sondern auch Einzelheiten der jeweiligen Architekturen vorhanden sein. Dieser Umstand macht es sehr schwer bzw. sehr aufwendig und teuer, effizient Programme zu erzeugen.

2.1.2.2 ILP-Techniken – Parallelität auf Instruktionsebene

Um die Leistungsfähigkeit von Prozessoren zu erhöhen, wird die **Parallelität auf Instruktionsebene** (ILP) ausgenutzt.

ILP – Instruction Level Parallelism

Definition 2.7 (CPI) Eine wichtige Maßzahl im Zusammenhang mit ILP stellt das Verhältnis zwischen der durchschnittlichen Anzahl von Zyklen pro Instruktion CPI bzw. dessen Kehrwert $IPC = \frac{1}{CPI}$ dar. Ein IPC Wert über 1 (bzw. $CPI < 1$) zeigt an, dass durchschnittlich IPC viele Instruktionen parallel ausgeführt werden. □

CPI – Cycles Per Instruction

IPC – Instructions Per Cycle

Pipelining ist eine ILP-Technik, die auf modernen Prozessoren eingesetzt wird [135], um mehrere aufeinander folgende Befehle des Programms parallel oder verschränkt abzuarbeiten. Dabei wird ein einzelner Befehl in seine Teilschritte zerlegt (s. Abschn. 2.1.2) und jeder Teilschritt innerhalb eines oder mehreren Prozessorzyklen nacheinander ausgeführt (auch als Pipeline bezeichnet). Die Anzahl von Teilschritten, die notwendig sind, um einen Befehl komplett auszuführen, wird als Tiefe der Pipeline bezeichnet. Da auf CISC-Prozessoren einzelne Instruktionen, aufgrund ihrer unterschiedlichen Komplexität, starke Unterschiede in den Ausführungszeiten aufweisen, werden die einzelnen komplexen Instruktionen in sogenannte **Mikro-Operationen** (μOps) aufgebrochen. Diese besitzen eine sehr ähnliche Ausführungszeit und können in einer Pipeline ausgeführt werden.

μOps – Micro Operations

Instruktion Nr.	**Pipeline Stufen**						
1	IF	ID	EX	WB			
2		IF	ID	EX	WB		
3			IF	ID	EX	WB	
4				IF	ID	EX	WB
Zyklus	1	2	3	4	5	6	7

Tabelle 2.1: Beispiel Pipeline-Ausführung

Beispiel 2.3: Pipeline-Ausführung

Tabelle 2.1 illustriert die Pipeline-Ausführung bzgl. der vier Phasen des Ausführungsmodells aus Abschnitt 2.1.2. Sie zeigt, dass im vierten Zyklus alle vier Befehle bearbeitet werden. Der erste Befehl steht in der Stufe vier des *Zurückschreibens – WB*. Der zweite Befehl steht in der Stufe der *Ausführung – EX*. Der dritte Befehl befindet sich in der *Dekodierung – ID* und der vierte Befehl wird gerade *geladen – IF*.

Pipelining verringert nicht die Zeit pro Instruktion, sondern erhöht vielmehr den Durchsatz an Instruktionen pro Zeiteinheit. Für eine Pipeline der Tiefe P und I ausgeführten Instruktionen, ergibt sich ein maximaler *SpeedUp* von:

$$SpeedUp = \frac{IP}{I + P - 1}$$

Im Beispiel 2.3 werden mit Pipelining sieben Prozessorzyklen zum Beenden aller vier Befehle benötigt, $IPC = \frac{4}{7}$. Ohne Pipelining sind es 16 Zyklen, $IPC = \frac{4}{16}$, das entspricht einem *SpeedUp* von $\approx 2,3$.

Für eine unendlichen Folge von Instruktionen gilt dann:

$$SpeedUp = \lim_{I \to \infty} \frac{IP}{I + P - 1} = P$$

Aus der Grenzwertbetrachtung ist zu schließen, dass sich der *SpeedUp* mit zu nehmender Tiefe der Pipeline erhöht. In der Praxis ist dieser Idealfall schwer zu erreichen, denn kann die nächste Instruktion nicht innerhalb des folgenden Zyklus ausgeführt werden, kommt es zu Problemen. Die Ursachen können in drei Gruppen unterteilt werden:

- Bei **strukturellen Ursachen** kann eine Instruktionskombination nicht innerhalb einer gegebenen Zyklenanzahl ausgeführt werden, da es der Hardware nicht möglich ist, bestimmte Instruktionskombinationen gleichzeitig auszuführen. Dies träfe beispielsweise für eine Hardwarekomponente zu, die zwei unterschiedliche Operationen zur gleichen Zeit ausführen soll.
- Bei **Datenproblemen** ist die Ursache darin zu suchen, dass benötigte Daten zum Zeitpunkt der Ausführung noch nicht zur Verfügung stehen (s. Bsp. 2.4). Bestimmte

Arten von Datenproblemen lassen sich durch **Umleiten** (engl. Bypassing, Forwarding) beseitigen. Dabei wird bereits nach dem *Ausführungsschritt* das Ergebnis für nachfolgende Instruktionen zur Verfügung gestellt.

- **Kontrollflussprobleme** entstehen, wenn Instruktionen (z.B. bedingte Sprünge) bereits in der Pipeline sind, die möglicherweise den Kontrollfluss, also den Programmzähler ändern. In diesem Fall kann der nächste Befehl nicht geladen werden, bis der Programmzähler die richtige Adresse enthält.

Alle drei Probleme können durch eine **Verzögerung** der Pipeline, auch als **Pipeline-Stall** oder **Bubble** bezeichnet, beseitigt werden. Bei einem Pipeline-Stall wird in die Pipeline solange die leere Instruktion (NOP) eingefügt, bis das Problem beseitigt ist.

NOP – No OPeration

Beispiel 2.4: Pipelining mit einem Datenproblem

In diesem Beispiel wird das Datenproblem als Ursache für einen Pipeline-Stall verdeutlicht. Es werden die folgenden beiden Instruktionen im vierstufigen Ausführungsmodell hintereinander in einer Pipeline der Tiefe vier abgearbeitet.

```
add REG1,7              // REG1 += 7    (1)
sub REG2,REG1           // REG2 -= REG1 (2)
```

Der Grund für den Pipeline-Stall liegt darin, dass beim Eintreten der Subtraktion in die Stufe der *Ausführung*, das Ergebnis der Additionsoperation sich in der Phase des *Zurückschreibens* befindet und somit noch nicht zur Verfügung steht.

Spekulative Ausführung ist eine weitere ILP-Technik. Dabei können einzelne Instruktionen bereits ausgeführt werden, ohne sicher zu sein, ob die Instruktionen überhaupt ausgeführt werden müssen. **Vorhersage** ist eine spezielle Form der spekulativen Ausführung und wird verwendet, um Kontrollflussprobleme zu beseitigen. Dazu wird eine Vorhersage über den Ausgang der Instruktion gemacht und mit dieser wird dann erst einmal weiter gerechnet. Im Fall, dass es sich um Sprünge im Programm handelt, wird von **Sprungvorhersage** gesprochen. Sobald die Instruktion, über die eine Vorhersage getroffen wurde, abgeschlossen ist, wird das Ergebnis mit der Vorhersage verglichen. War die Vorhersage korrekt, wird weiter gearbeitet. Im Fall einer falschen Vorhersage muss die Pipeline für ungültig erklärt werden (engl. Pipelineflush) und die Abarbeitung beginnt von neuem ab der Stelle, an der die Vorhersage falsch war.

Ein anderes Problem beim Pipelining ist die Blockierung der Pipeline, wie sie z.B. durch das Warten auf Speicherzugriffe verursacht wird. Werden Instruktionen **in vordefinierter Reihenfolge** (InO) ausgeführt, müssen alle nachfolgenden Instruktionen auf die Beendigung des Speicherzugriffs warten. Um dieses Problem zu umgehen, wird versucht, nachstehende Instruktionen zu finden, die keine Abhängigkeit auf die wartende Instruktion haben und diese dann **außerhalb der vordefinierten Reihenfolge** (OoO) auszuführen.

InO – In-Order

OoO – Out-of-Order

OoO-Ausführung ist eine ILP-Technik, bei der einzelne Instruktionen unabhängig voneinander ausgeführt werden. Um die Korrektheit der Ausführung zu garantieren und trotzdem möglichst viele voneinander unabhängige Instruktionen zu finden, sind hoch komplexe Prozessoren bzw. Befehlssteuerungen notwendig. Diese nehmen eine Vielzahl der Transistoren auf dem Prozessorchip ein. Daher sind InO-Prozessoren deutlich kleiner[3] und preiswerter als OoO-Prozessoren.

Superskalare Ausführung ist die letzte ILP-Technik, die in dieser Arbeit vorgestellt wird. Darin werden einzelne zur Ausführung gehörende Teile des Prozessors (z.B. ALU, Teile der Pipeline) repliziert, damit mehrere Instruktionen oder Pipelines parallel ausgeführt werden können, ohne strukturelle Probleme zu verursachen.

SIMD – Single Instruction Multiple Data

Definition 2.8 (SIMD) Bei der **SIMD**-Befehlsausführung wird ein einzelner Befehl gleichzeitig auf mehreren Argumenten bzw. Daten ausgeführt. □

SIMD stellt eine tatsächliche Datenparallelität in der Instruktionsausführung bereit. Der Grad der Parallelität hängt dabei zum einen von der SIMD-Befehlsbreite ($|SIMD|$) ab. Diese gibt dabei an, wie viele Bits mit einem Befehl gleichzeitig verarbeitet werden können, d.h., wie groß die einzelnen SIMD-Register sind. Zum anderen hängt der Grad vom Datentyp der Argumente ab, so können bei einer SIMD-Breite von 128 Bit 16 8 Bit-Argumente oder 8 16 Bit-Argumente usw. gleichzeitig mit einem einzigen Befehl verarbeitet werden. Dazu ist es notwendig, dass der Programmierer die entsprechenden Befehle kennt und verwendet. SIMD-Befehle sind meist in speziellen Erweiterungen zur jeweiligen Prozessor-ISA definiert und somit extrem von der jeweiligen Rechnerarchitektur abhängig.

Im Folgenden soll die Verwendung von SIMD-Operationen an einem Beispiel vorgestellt werden, später wird diese Arbeit im Abschnitt 3.4 speziell auf die Verwendung von SIMD-Operationen eingehen. Der Quelltext im Beispiel 2.5 soll beispielhaft die Verwendung von SIMD-Operationen zur Ausführung eines parallelen Vergleichs darstellen. Abbildung 2.6 illustriert die Inhalte der einzelnen Variablen, sowie die Vorgänge bei den SIMD-Operationen.

Beispiel 2.5: SIMD-Operationen zum parallelen Vergleich

```
1 size_t convMoveMask_Shift(uint32_t mask)
2 {
3     for(size_t i=0;i<4;++i)
4     {
5         p += mask & 0x01;
6         mask >>= 4;
7     }
8     return (4 - p);
9 }
10
11 size_t convMoveMask_PopCnt(uint32_t mask)
12 { return (4 - __popcnt(mask)/4); }
```

[3] weniger Transistoren daher weniger Fläche

```
13
14 size_t convMoveMask_Case(uint32_t mask)
15 {
16     size_t p = 4;
17     switch(mask)
18     {
19     case 0xffff: p = 0; break;
20     case 0xfff0: p = 1; break;
21     case 0xff00: p = 2; break;
22     case 0xf000: p = 3; break;
23     }
24     return p;
25 }
26
27 size_t compareKeys(int32_t* vals, int32_t key)
28 {
29     __m128i key128  = _mm_set1_epi32 (key);
30     __m128i vals128 = _mm_load_si128 ((__m128i*) vals);
31     __m128i cmp128  = _mm_cmpgt_epi32 (vals128, key128);
32     uint32_t mask   = _mm_movemask_epi8 (cmp128);
33     return convMoveMask_...(mask);
34 }
35
36 void sample()
37 {
38     int32_t list[] = { 1, 3, 8, 10};
39     int32_t key = 9;
40     size_t pos = compareKeys(list, key);
41     if(pos > 0)
42         cout << list[pos-1] << "<=";
43     cout << key;
44     if(pos < 4)
45         cout << "<" << list[pos];
46 }
```

Im Beispiel soll eine sortierte Liste von vier 32 Bit-Zahlen $L = (l_0, l_1, l_2, l_3)$ mit dem Schlüssel k verglichen und seine Position p ($0 \leq p \leq 4$) bestimmt werden, so dass gilt:

$$\forall l_i \text{ gilt } \begin{cases} l_i < k & \text{if } 0 \leq i < p-1 \\ l_i \leq k & \text{if } i = p-1 \\ l_i > k & \text{if } p \leq i \leq 4. \end{cases}$$

Dazu wird zunächst der Schlüssel k mit Hilfe des Befehls _mm_set1_epi32 gleichzeitig in alle vier 32 Bit-Segmente eines SIMD-Registers geladen (Zeile 29, Abb. 2.6 rechts oben). Anschließend werden die vier Elemente der Liste gleichzeitig über den Befehl _mm_load_si128 in ein zweites SIMD-Register geladen (Zeile 30, Abb. 2.6 links oben). Darauf folgt der paarweise Vergleich der Segmente (Zeile 31). Das Ergebnis ist -1 (*0xFFFFFFFF*) im entsprechenden Segment, falls der Listenwert größer als der Schlüs-

sel ist; anderenfalls wird das Segment mit 0 gefüllt (Abb. 2.6 mitte). Um das Ergebnis des Vergleichs zu extrahieren, wird aus dem Ergebnis-SIMD-Register durch den Befehl _mm_movemask_epi8 von jedem 8 Bit-Segment das höchstwertige Bit extrahiert und an der entsprechenden Stelle in einer 32 Bit-Zahl gespeichert (Zeile 32, Abb. 2.6 unten). Im Beispiel wird der Wert *0xF000* zurückgeliefert. Diese Bitmaske kann dann durch eine der *convMoveMask* Methoden interpretiert und die korrekte Position für den Vergleich des Schlüssels mit der Liste ermittelt werden. Im Beispiel ist die Rückgabe die Position 3 und die Ausgabe würde lauten:

$$8 <= 9 < 10$$

Wie zu sehen ist, kann die Evaluierung der Bitmaske auf unterschiedliche Art und Weise erfolgen. Im Beispiel sind dazu die Möglichkeit des Bitschiebens (*convMoveMask_Shift*), der Verwendung einer Case-Anweisung (*convMoveMask_Case*) und der Nutzung eines speziellen Befehl *popcnt* (*convMoveMask_PopCnt*) angegeben, wobei der Befehl *popcnt* die Anzahl der gesetzten Bits zurückgibt. Im Abschnitt 3.4 werden die drei Methoden bzgl. ihrer Leistungsfähigkeit evaluiert.

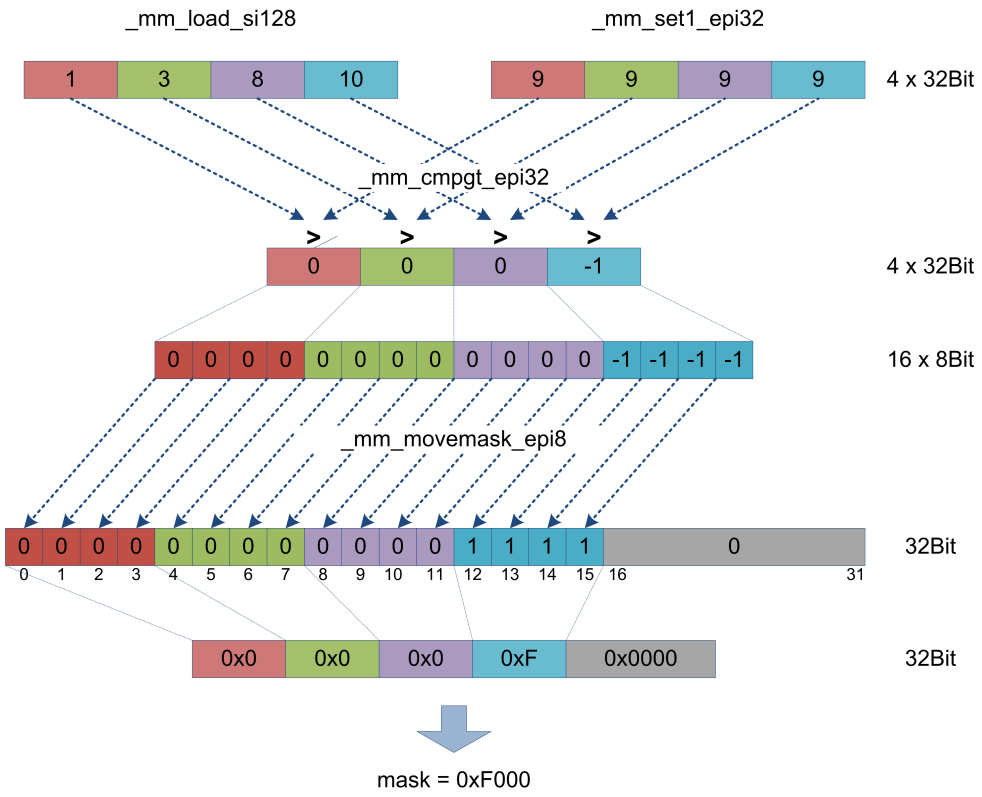


Abbildung 2.6: Darstellung zum Beispiel 2.5

Fazit: Die Nutzung von ILP kann zu einer enormen Leistungssteigerung führen. Allerdings ist dazu erforderlich, dass die Instruktionen eines Programms tatsächlich zu einem hohen Prozentsatz diese Arten von Parallelitäten aufweisen. Daher sind bei Programmen mit einem hohem Anteil an Kontrollkonstrukten oder Speicherzugriffen Grenzen in der Nutzung von ILP gesetzt. Bei ihrer Überschreitung nimmt die Leistung nicht mehr zu, sondern bleibt gleich bzw. nimmt sogar ab. Wird beispielsweise die Tiefe einer Pipeline erhöht, führt diese Erhöhung auch zu einer Erhöhung von notwendigen Vorhersagen und damit zu einer Erhöhung der Falschvorhersagen, was wiederum ein vermehrtes löschen der Pipeline zur Folge hat. Aktuellen Rechnerarchitekturen besitzen Pipelines mit einer Tiefe von bis zu 31. Eine weitere Erhöhung der Anzahl der Pipelinetiefe wird aus den zuvor genannten Gründen als nicht sinnvoll betrachtet [82]. Die SIMD-Breite steigt dagegen weiter an und wird mit der nächsten Intel-Mikroarchitektur auf 256 Bit[4] wachsen [50]. SIMD bietet somit einen starken Leistungsgewinn, dazu ist es erforderlich, dass der Programmierer die entsprechenden Befehle verwendet oder sein Programm entsprechend schreibt. Im Abschnitt 3.4 des dritten Kapitels wird ein Index vorgestellt, der die Vorteile der SIMD-Verarbeitung nutzt und damit einen signifikanten Leistungsgewinn erreicht.

2.1.2.3 Mehrprozessor-Computer

Mehrprozessor-Computer besitzen eine Shared-Memory-Architektur (s. Abschn. 1.1.3), bei der mehrere Prozessoren in einem einzelnen Computersystem betrieben werden. Sie können in Bezug auf ihren Umgang mit den Prozessoren wie folgt unterschieden werden: (i) sind alle Prozessoren gleichartig und werden auch gleich behandelt, wird von **symmetrischen Mehrprozessoren** (SMP) gesprochen. SMP verbindet alle Prozessoren über einen oder mehrere Systembusse (s. Abb. 2.7(a)) mit dem gesamten gemeinsamen Speicher; die Kosten für den Zugriff auf den Speicher sind für jeden Prozessor gleich.

SMP – Symmetric Multi Processing

Der Nachteil von zentralen busbasierten Systemen wie SMP ist, dass der Bus selbst zur Engstelle wird und somit SMP-Systeme keine gute Skalierung für eine große Anzahl von Prozessoren bieten [79]. Den Gegensatz zu SMP stellt (ii) der nicht einheitliche Speicherzugriff (NUMA) dar.

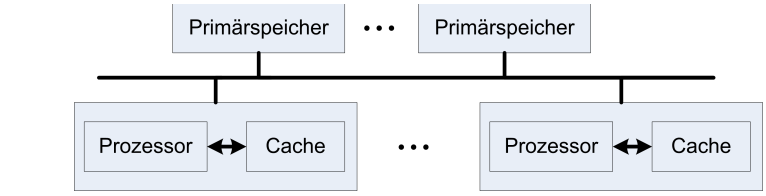
NUMA – Non-Uniform Memory Access

Definition 2.9 (NUMA) NUMA ist eine Architekturform für Mehrprozessor-Computern, bei der Bereiche des physischen Speichers an einzelne Prozessoren gekoppelt wird und die Einheiten von Prozessoren und prozessor-lokalen Speicher über ein Netzwerk miteinander verbunden werden (s. Abb. 2.7(b)) [79]. □

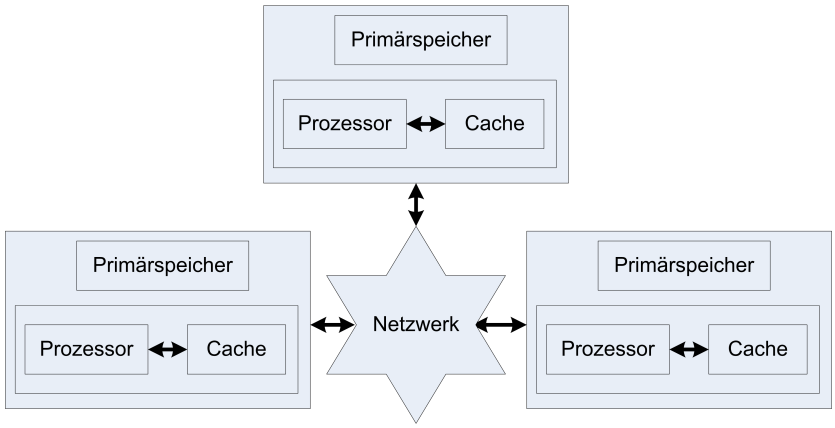
In einem NUMA-System kann der Zugriff auf Speicherbereiche unterschiedlich teuer sein, da für einen nicht prozessor-lokalen Speicherzugriff andere Prozessor-Speicher-Einheiten angesprochen werden müssen. Diese nicht lokalen Zugriffe bedeuten zusätzliches Warten, in Abhängigkeit von der Erreichbarkeit, Bandbreite und Auslastung der jeweiligen Einheiten und des Netzwerks. Eine Spezialform von NUMA ist das ccNUMA.

ccNUMA – Cache Coherent NUMA

[4] 512 Bit für den Intel-Larrabee



(a) SMP



(b) NUMA

Abbildung 2.7: Mehrprozessor Computer

Beim ccNUMA sowie auch bei SMP werden Cache-Kohärenz-Protokolle (s. Abschn. 2.1.1.2) eingesetzt. NUMA- bzw. ccNUMA-Architekturen werden als die zukünftige Architektur für Mehrprozessor-Systeme angesehen, da sie sich im Gegensatz zu SMP auch für eine hohe Anzahl von Prozessoren verwenden lassen [99].

2.1.3 Der Intel-Pentium-4-Prozessor

Im Folgenden werden die zuvor eingeführten Konzepte aus den Abschnitten 2.1.1 und 2.1.2 am Beispiel des Intel-Pentium-4-Prozessor (P4) vorgestellt. Abgesehen vom Bussystem stellt der P4 in einigen Bereichen ein Maximum in Bezug auf das Ausreizen der Möglichkeiten der Leistungssteigerung dar. Er ist damit in Bezug auf interne Prozessorfrequenz und Nutzung von ILP durch Pipelining und OoO-Ausführung heutigen Prozessorarchitekturen nahezu gleichwertig und soll daher als Anschauungsobjekt dienen.

Der P4 – im November 2000 erstmals präsentiert – folgt der Intel-Net-Burst-Mikroarchitektur und implementiert die ISA IA-32 [82]. Abbildung 2.9(a) zeigt eine Mikrofotografieaufnahme des P4, dazu stellen die Abbildungen 2.8, 2.9(b) und 2.9(c) die einzelnen Bereiche, ihre Funktionen und Zusammenhänge dar. Anfänglich wurde der P4 in eine Strukturgröße von $0,18\mu m$ gefertigt, damit war er $217mm^2$ groß, bestand aus 42

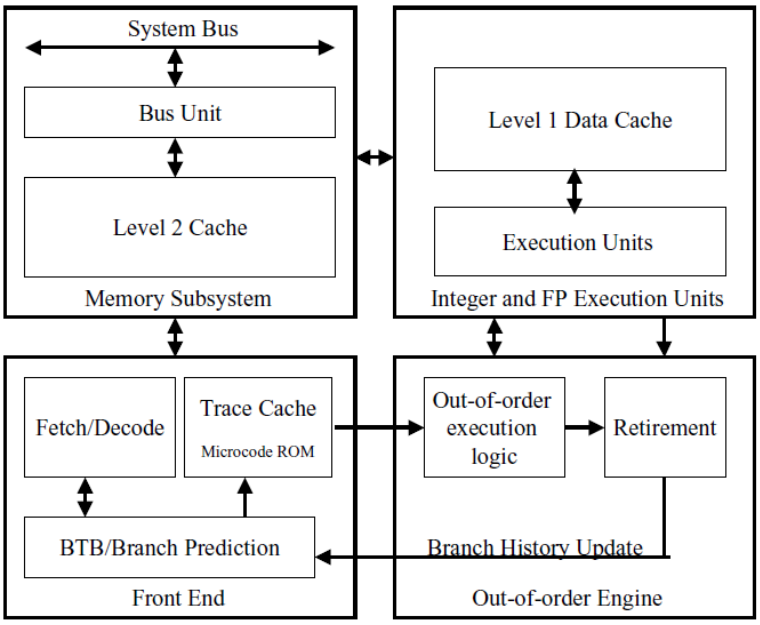


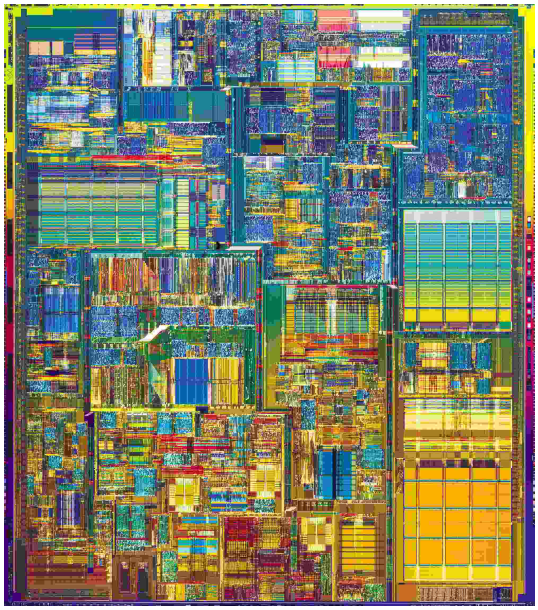
Abbildung 2.8: Blockdiagram eines Intel-Pentium-4-Prozessors [82]

Millionen Transistoren und lief bei einer Taktfrequenz[5] von $1,5GHz$. Er besitzt zwei Cache-Stufen mit eine Zeilenbreite von 128 Byte[6]. Der L1-Data-Cache ist 8 KByte groß und 4-Wege assoziativ; der L2 ist 256 KByte groß und 8-Wege assoziativ. Die SIMD-Befehlsbreite liegt bei 128 Bit.

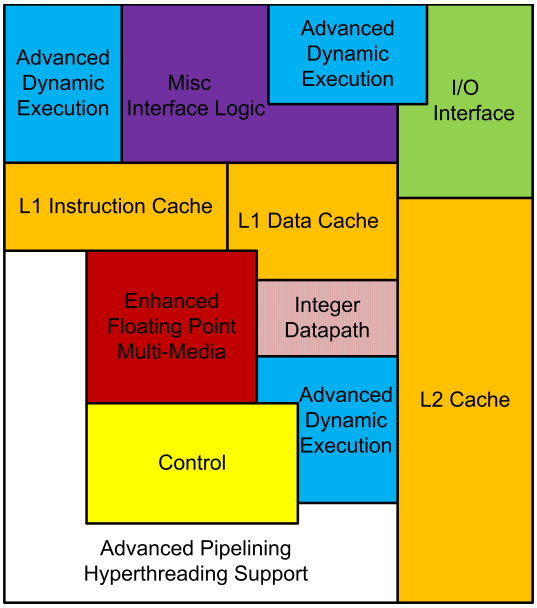
Der P4 lädt IA-32-Instruktionen und dekodiert sie durch das InO-Front-End (Abb. 2.8 unten links, Abb. 2.9(c) der obere linke Bereich) in Sequenzen von μOps. Um diesen Prozess zu beschleunigen, besitzt der P4 einen sogenannten *Execution-Trace-Cache*. Dieser Cache ist eine spezielle Form eines L1-Instruktionscaches und kann nicht nur IA-32-Instruktionen speichern, sondern auch die dekodierte μOp Sequenz der Instruktion (bis zu 12K μOps). Falls eine Instruktion nicht im Trace-Cache gefunden wird, muss sie aus dem L2-Cache bzw. aus dem Primärspeicher über den L2-Cache gelesen werden. Die Planung der Ausführung der einzelnen μOps wird dann durch die OoO-Engine (Abb. 2.8 unten rechts, Abb. 2.9(c) der mittlere Bereich) übernommen. Die eigentliche Ausführung übernehmen die ALUs (Integer and Floating Point (FP) Units – Abb. 2.8 oben rechts bzw. Abb. 2.9(c) der untere Bereich).

Das Speichersystem (memory subsystem – Abb. 2.8 oben links, Abb. 2.9(c) der äusserst rechte Bereich) verbindet den Prozessor mit dem Primärspeicher. Es umfasst den L2-Cache sowie den Systembus (FrontSideBus – FSB), der Daten aus dem Primärspeicher lädt bzw. Daten in den Primärspeicher schreibt. Er ist 64 Bit breit, wird mit einer Taktfrequenz von $100MHz$ betrieben und kann synchron bis zu vier Datenelemente übertragen. Das entspricht einer maximalen Datentransferrate von 3 GByte pro Sekun-

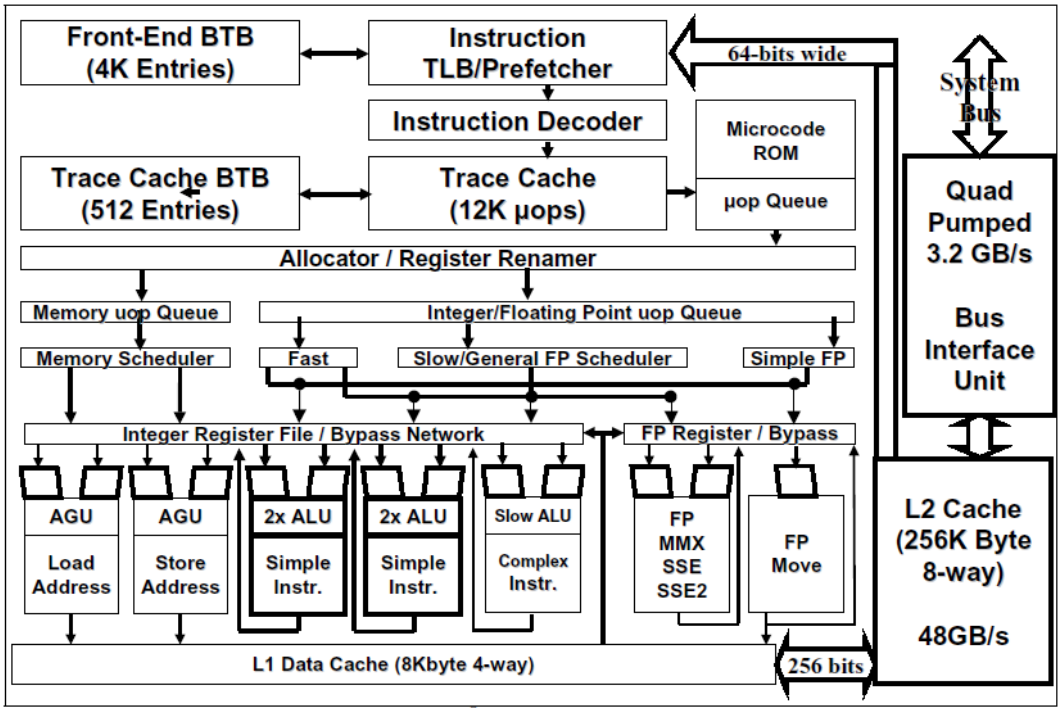
[5] die ALU bei doppelter Frequenz
[6] unterteilt in zwei 64 Byte Blöcke



(a) Mikrofotografie [91]



(b) Funktionseinheiten



(c) Mikroarchitektur [82]

Abbildung 2.9: Intel-Pentium-4-Prozessor

de. Der P4 konnte in einem Mehrprozessor-System mit bis zu vier Prozessoren verwendet werden, bei dem die Prozessoren gemeinsam über den FSB kommunizieren (s. Abb. 2.10).

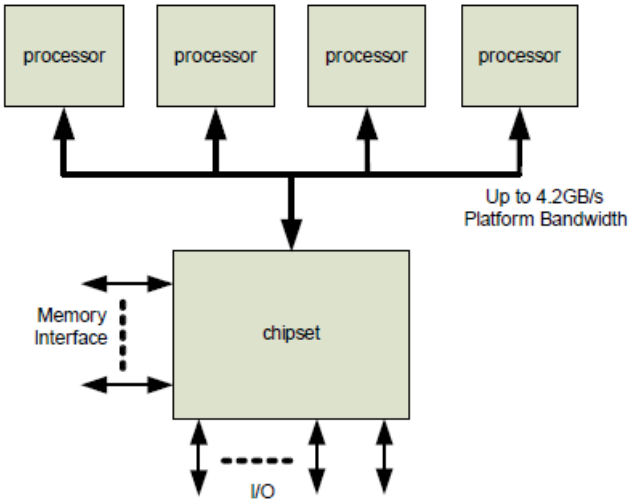


Abbildung 2.10: Der FSB des P4 [99]

Der FSB hat nach Definition 2.6 einen maximalen *SR* Wert von 6 (s. Bsp. 2.2). Nach Spracklen et al. [152] erzeugen moderne OoO-Prozessoren etwa zwei simultane Anfragen auf einer typischen Serverarbeitslast, damit war der FSB bereits mit vier Prozessoren nahezu ausgelastet oder sogar überlastet und wurde oft zu einem Flaschenhals [173, 30].

2.1.4 Das Mooresche Gesetz

Wie bereits mehrfach erklärt, sind Prozessoren intern aus Transistoren aufgebaut und ihre Anzahl kann als ein ungefähres Maß für die Komplexität und Leistungsfähigkeit des Prozessors betrachtet werden. Gorden Moore, Mitgründer der Intel-Corporation, stellte in den sechziger Jahren eine Behauptung auf, die heute als **Mooresche Gesetz** bekannt ist [126]. Dieses Gesetz besagt, dass sich alle 18 bis 24 Monate die Anzahl der Transistoren auf einem IC verdoppeln lässt. Diese Verdoppelung wird durch eine Verkleinerung der Strukturen erreicht.

Abbildung 2.11 stellt die Entwicklung verschiedener Merkmale von Prozessoren über die letzten 35 Jahre dar, darunter: (i) Die Anzahl der Transistoren pro Prozessor, (ii) die interne Prozessorfrequenz, (iii) die Taktfrequenz des Systembusses und (iv) der Kehrwert des Herstellungsprozesses (Strukturgröße oder Prozessgröße). Das Diagramm verdeutlicht, dass das Mooresche Gesetz noch immer Bestand hat und die Entwicklung auch noch mindestens die nächsten zehn Jahre korrekt widerspiegelt [136]. Das Mooresche Gesetz ist nicht auf Prozessoren begrenzt, vielmehr entwickeln sich auf ICs beruhenden Teile eines Computers in diesem Sinne. So zeigt z.B. die Entwicklung der Kapazität von DRAM eine Vervierfachung alle drei Jahre [135].

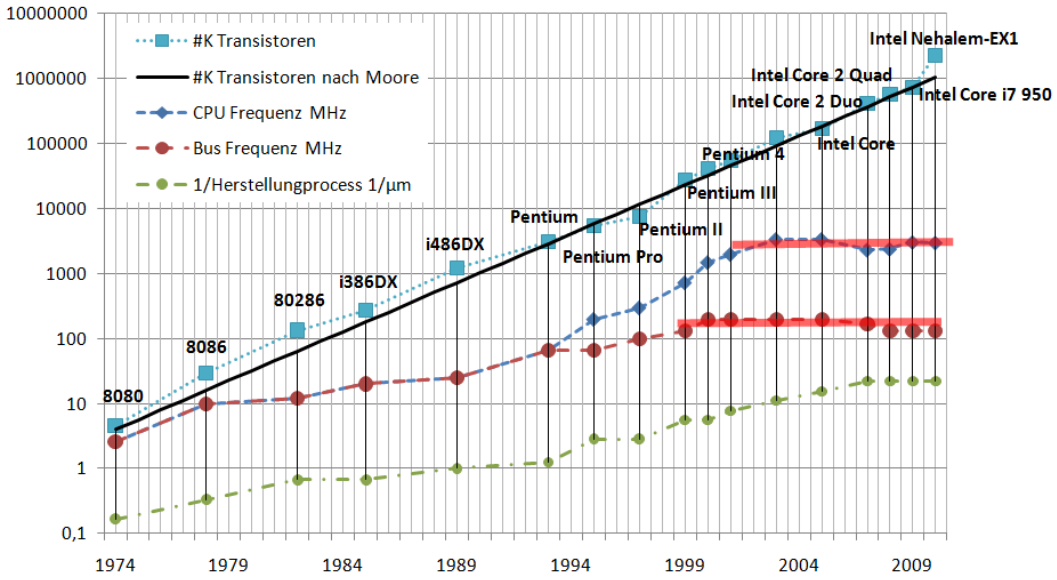


Abbildung 2.11: Zeitlicher Verlauf der Hardwareentwicklung (Basisdaten aus [94])

Bei Prozessoren wurde die Verkleinerung der Strukturgröße in der Vergangenheit hauptsächlich für zwei Dinge genutzt. Zum einen wurde eine stetige Steigerung der Taktraten erreicht und damit ein automatischer Leistungsgewinn der Nachfolgergeneration über ihren Vorgänger erzielt; zum anderen wurden die zusätzlichen Kapazitäten an Transistoren für unterschiedliche Erweiterungen genutzt. Darunter fallen z.B. die Änderung der internen Wortbreite von 8 Bit auf heute 64 Bit, immer größere Caches und Busbreiten. Des Weiteren wurden zusätzliche Funktionalitäten implementiert, dazu gehören interne Pipelineverarbeitung, OoO-Ausführung, SIMD-Operationen für bis zu 256 Bit große Operanden und die Replikation von Funktionseinheiten wie der ALU, um Maschinenbefehle parallel zueinander abzuarbeiten [142].

Insbesondere der stetige Frequenzanstieg und die Nutzung von ILP-Techniken haben in der Vergangenheit zu einem Leistungsgewinn von der Vorgänger- zur Nachfolger-Prozessorgeneration geführt. Er hat ebenfalls für Software einen automatischen Leistungsgewinn bedeutet, ohne das irgendetwas an der Software verändert werden musste und wird daher oft als „Free Lunch“ oder „Free Ride“ bezeichnet [153, 119, 118]. Abbildung 2.11 zeigt die stetige Steigerung der internen Prozessorfrequenz, bis etwa um 2001 mit dem P4 ein Maximum erreicht wurde. Seit diesem Zeitpunkt hat sich die Frequenz bei etwa $3GHz$ eingependelt, was wie folgt zu begründen ist:

1. Durch die stetige Erhöhung der Prozessorfrequenz entstand ein enormer Latenzunterschied zwischen dem Primärspeicher und der CPU bzw. den CPU-Internen-Caches; dieses Phänomen wird als **Memory-Wall** bezeichnet [119]. Der Prozessor muss auf Daten, die nicht in den internen Caches vorhanden sind, bis zu mehrere hundert Zyklen warten. Eine weitere Erhöhung der Frequenz würde daher kontraproduktiv sein, denn obwohl einzelne Befehle schneller abgearbeitet würden,

müsste auf nicht vorhandene Daten länger (mehr Zyklen) gewartet werden. Dieses Missverhältnis wurde im Datenbankumfeld von Ailamaki et al. [8] untersucht.

2. Mit der Erhöhung der Frequenz geht eine erhöhte Leistungsaufnahme einher, welcher physikalische und wirtschaftliche Grenzen gesetzt sind. Dieses Problem wird als **Power-Wall** bezeichnet [119] und kann anhand der Definition der Leistungsaufnahme P erklärt werden (s. Bsp. 2.6):

$$P = nCV^2 f + leakage$$

Dabei sind: C die Kapazität der Transistoren, V die Spannung, f die Frequenz, n die Gesamtanzahl der Transistoren auf dem Prozessorschaltkreis und *leakage* die Verlustleistung. Die Herausforderung ist zum einen die benötigte **Leistung** auf den Schaltkreis zu leiten und zum anderen sie – in Form von Wärme – auch wieder vom Schaltkreis abzuführen. Dabei bedeutet die Verkleinerung der Strukturen gleichzeitig eine Erhöhung der Wärme pro Flächeneinheit.

3. Mit wachsender Anzahl von Transistoren, wächst auch die Prozessor interne Leitungslänge für den Signaltransport [142, 164]. Ein Signal kann bei einer Frequenz von $3GHz$ – also in $0,33ns$ – im Vakuum[7] einen Weg von $0,1m$ zurücklegen. Die Lichtgeschwindigkeit im Medium und somit in Schaltkreisen ist deutlich kleiner und damit werden die Ausmaße und die Signalwege ein beschränkender Faktor.

Beispiel 2.6: Erhöhung der Leistungsaufnahme

Vergleicht man die Prozessorentwicklung aus dem Jahr 2008 mit der für das Jahr 2016, so wird die Anzahl der Transistoren nach dem Gesetz von Moore etwa 16 mal größer sein als acht Jahre zuvor. Die Spannung wird leicht von $1,0$ Volt auf $0,7$ Volt fallen, ebenso wie die Kapazität [119]. Daraus folgt für die Leistungsaufnahme bei gleicher Frequenz, ein Wachstum um einen Faktor sechs bis acht.

Fazit: Eine weitere Steigerung der Leistung ist allein durch Frequenzerhöhung kaum mehr möglich und andere Methoden zur Leistungssteigerung wie die Nutzung von ILP-Techniken durch Pipelining und OoO-Ausführung sind nur bis zu einem bestimmten Grad sinnvoll und stark von dem ausgeführten Programm abhängig. Um die Leistungsfähigkeit zukünftiger Prozessoren zu steigern, wurde daher dazu übergegangen, immer größere Teile des Prozessors zu replizieren und hat letztendlich dazu geführt, dass mehrere Kerne auf einen Prozessor untergebracht werden.

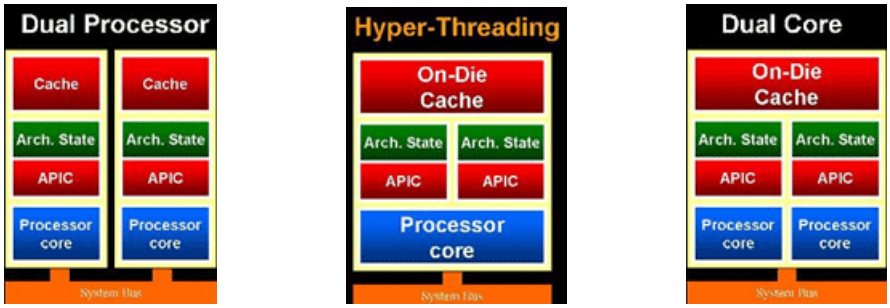
[7] Lichtgeschwindigkeit $c = 299792458 \frac{m}{s}$

2.1.5 Mehrkern-Prozessoren

Definition 2.10 (Kern) Eine Verarbeitungseinheit, auch **Kern** (engl. Core) genannt, beschreibt alle für die Instruktionsverarbeitung notwendigen Bestandteile eines Prozessor. Dazu zählen der Programmzähler, der Instruktionsspeicher (L1-Instruktion-Cache), die Register, die ALU sowie ein Datenspeicher (L1-Daten-Cache). Erweiterungen wie der L2-Cache und das Speichersystem gehören nicht direkt zum Kern und werden oft unter dem Begriff **Uncore** zusammengefasst [18]. □

Bisher wurde der Begriff des Prozessors genutzt, um auf eine **einzelne Verarbeitungseinheit** also einen einzelnen Kern zu verweisen. Dies gilt auch für die im Abschnitt 2.1.2.3 vorgestellten Mehrprozessor-Systeme. Der Einzelkern-Prozessor kann zu einem Zeitpunkt nur ein Programm ausführen, da er nur einen einzigen *Programmzähler* besitzt. Um mehrere Programme nebenläufig (s. Def. 2.17) auf dem Prozessor auszuführen, muss das Betriebssystem (OS) die einzelnen Programme von Zeit zu Zeit in ihrer Abarbeitung anhalten, ihren aktuellen Status (Register, Programmzähler) im Primärspeicher ablegen und den Status eines anderen Programms laden, welches dann für einen bestimmten Zeitraum weiterarbeiten kann. Abbildung 2.12(a) zeigt ein Dual-Prozessor-System, mit zwei über einen Bus verbundenen Einzelkern-Prozessoren.

OS – Operating System



(a) Dual-Prozessor-System

(b) SMT-Prozessor

(c) Dual-Core-Prozessor – CMP

Abbildung 2.12: Vergleich von Mehrkern-Prozessoren mit Mehrprozessor Systemen [116]

2.1.5.1 Architekturtypen von Mehrkern-Prozessoren

Definition 2.11 (Mehrkern-Prozessor) **Mehrkern-Prozessoren** sind Prozessoren, die mehrere Programme oder Instruktionspfade parallel abarbeiten können. Dazu werden bestimmte Bereiche des Prozessors repliziert oder gemeinsam verwendet. Auf Basis der gemeinsamen Nutzung oder der Replikation, existieren für Mehrkern-Prozessoren drei Architekturtypen [152, 116]. □

In einem superskalaren Einzelkern-Prozessor kommt es oft vor, dass die vorhandenen ALUs nicht optimal genutzt werden, da sich in Programmen zu wenig voneinander unabhängige Instruktionen finden lassen, so dass die Pipeline durch lange Speicherzugriffe

immer wieder angehalten werden muss. Um die vorhandene Hardware besser zu nutzen, ist eine mögliche Lösung, zwei oder mehr Programme gleichzeitig ablaufen zu lassen.

Definition 2.12 (SMT) Bei einer als **simultanen Multithreading** (SMT, HT) bezeichneten Prozessorarchitektur werden nur die Prozessorbereiche dupliziert, die den Prozessorzustand darstellen (Programmzähler, Register, Interruptkontroller), während alle anderen Hardwarekomponenten (z.B. Caches, Fließkommaeinheiten und Speicherbusanbindungen) des Prozessor gemeinsam genutzt werden. Es werden somit mehrere **logische** Prozessoren durch die Hardware simuliert. □

SMT – Simultaneous Multi Threading

HT – Hyper-Threading

Der Vorteil von SMT ist, dass durch gleichzeitige Abarbeitung mehrerer Instruktionsströme die Möglichkeit besteht, mehr voneinander unabhängige Instruktionen zu finden. Abbildung 2.12(b) stellt die gemeinsamen bzw. getrennten Teile eines SMT-Prozessors dar. Seilter et al. merken in [149] an, dass die Leistung des Kerns auf Grund von Überbelastungen des Caches und des Adressübersetzungsspeichers (TLB) eingeschränkt wird, je mehr Threads auf dem gleichen Kern unabhängige Aufgaben ausführen. Der P4 wurde in späteren Versionen um das Hyper-Threading-Konzept erweitert und konnte so zwei Befehlsströme gleichzeitig abarbeiten. Der durchschnittliche Leistungsgewinn bei der Nutzung von SMT liegt zwischen 20 und 30% [95, 26]. Eine andere Möglichkeit ist es, den kompletten Kern zu replizieren und mehrere vollständig unabhängige Kerne in einem Prozessor unterzubringen.

TLB – Translation Lookaside Buffer

Definition 2.13 (CMP) Eine Prozessorarchitektur wird mit **Mehrprozessor auf Chipebene** (CMP) bezeichnet, wenn mehrere Kerne auf einem Prozessor untergebracht sind [152]. CMP-Prozessoren können durchaus aus einer Vielzahl von unterschiedlichen Kernen bestehen [80, 18, 159]. Hat ein Prozessor acht oder weniger Kerne spricht man von **Multi-Core**, im Fall von mehr als acht Kernen von **Many-Core** [118, 3]. □

CMP – Chip-level MultiProcessing

Ein CMP-Prozessor ist einem SMP-System sehr ähnlich; Unterschiede sind allerdings darin festzustellen, dass alle Prozessoren auf einem Chip untergebracht sind und die Uncore-Bestandteile (s. Def. 2.10) gemeinsam nutzen. Ein weiterer wichtiger Vorteil ist, dass die Kommunikation zwischen den Kernen auf einem Prozessorchip deutlich schneller ist als über einen externen Bus. Abbildung 2.12(c) illustriert einen Dual-Core-Prozessor, dessen Kerne sich einen gemeinsamen Cache teilen. Abbildung 2.13 zeigt eine Mikrofotografie eines Intel-Quad-Core-Prozessors, bei der deutlich erkennbar ist, dass es sich um zwei Dual-Cores Dies handelt, die in einem Gehäuse untergebracht sind.

Definition 2.14 (CMT) Multithreading auf Chipebene (CMT) ist die Kombination von SMT- und CMP-Architekturen. Dabei werden mehrere SMT-Kerne in einem Prozessor untergebracht [152]. □

CMT – Chip-level MultiThreading

Definition 2.15 (Hardware-Thread) Ein **Hardware-Thread** ist die kleinste Einheit, die in der Lage ist, ein Programm auszuführen, dabei bezieht sich diese Bezeichnung allerdings auch auf einzelne Kerne bzw. durch SMT geteilte Kerne. □

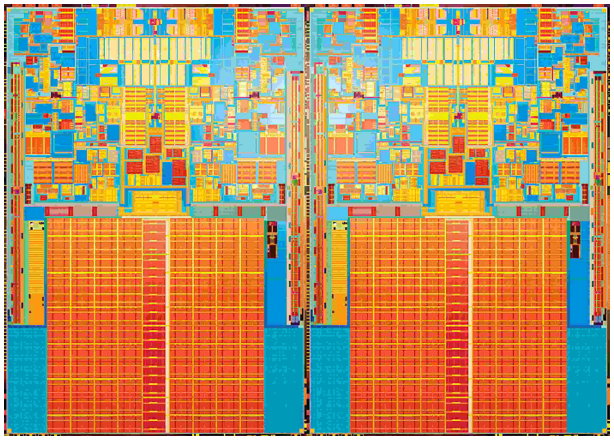


Abbildung 2.13: Mikrofotografie eines Intel-Quad-Core [96]

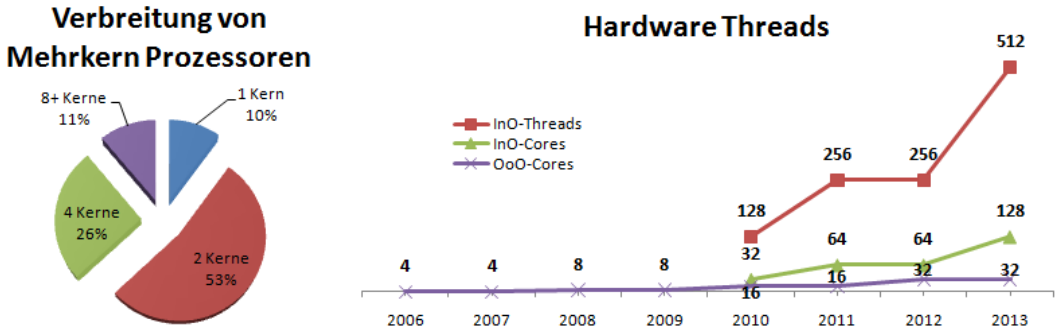
Ein Prozessor mit z.B. zwei Kernen, die je vier logische Prozessoren durch SMT simulieren, umfasst somit insgesamt acht Hardware-Threads. Bezüglich Anordnung und Verbindung einzelner Kerne werden drei Modelle unterschieden [142]. Beim (i) **hierarchischen Modell** werden mehrere Kerne durch einen Cache bzw. eine Cachehierarchie miteinander verbunden. Vom (ii) **Pipeline-Modell** spricht man, wenn Daten schrittweise durch mehrere Kerne verarbeitet werden, wie es beispielsweise bei Grafikprozessoren üblich ist. Das letzte Modell, ist das (iii) **netzwerkbasierte Design**, dabei hat jeder Kern einen eigenen lokalen Cache und die Kerne sind über ein Netzwerk miteinander verbunden.

2.1.5.2 Verbreitung und Entwicklung von Mehrkern-Prozessoren

Im Januar 2009 wurden die Ergebnisse einer Umfrage veröffentlicht, die die Verteilung von Einzelkern- und Mehrkern-Prozessoren untersucht [124]. Die Abbildung 2.14(a) zeigt eine Dominanz von Dual- und Quad-Core-Prozessoren und verifiziert die Aussage von Azimi et al. [18] aus dem Jahr 2007.

Abbildung 2.14(b) zeigt eine geschätzte Entwicklung bzgl. der Anzahl von Kernen von Herb Sutter [159]. Demnach besteht im Jahr 2013 ein OoO-Prozessor aus bis zu 32 Kernen, ein InO-Prozessor aus bis zu 128 Kernen. Weiter geht er davon aus, dass InO-Prozessoren der CMT-Architektur (Def. 2.14) folgen und jeder Kern bis vier logische Prozessoren durch SMT simulieren kann und somit ein einzelner InO-Prozessor bis zu 512 Hardware-Threads besitzt.

Solche Mehrkern-Prozessoren stellen enorme Herausforderungen an das I/O- und Speichersystem [18, 136]. Nach Pawloski [136] liegt die notwendige Bandbreite eines Prozessors bei etwa 1 Byte pro Sekunde pro Operation, was bei einer Taktfrequenz von $3GHz$ und einem Mehrkern-Prozessor mit 1000 Kernen zu einer notwendige Bandbreite von etwa 3 TByte pro Sekunde führt. Weiter behaupten Azimi et al. in [18], dass heutige Cache-Kohärenz-Protokolle bei einer hohen Anzahl von Kernen die Leistungsfähigkeit



(a) Verbreitung von Mehrkern-Prozessoren

(b) Anzahl von Kernen und Hardware-Threads

Abbildung 2.14: Verbreitung und Entwicklung von Mehrkern-Prozessoren

stark beschränken und somit neue Protokolle oder der Verzicht auf diese Protokolle notwendig werden.

2.1.5.3 Aktuelle und zukünftige Mehrkern-Prozessoren

Im Folgenden werden Mehrkern-Prozessoren und ihre Eigenschaften vorgestellt. Die Tabelle 2.2 listet die einzelnen Prozessoren mit der Anzahl der Kerne, der Anzahl der Transistoren, den Caches und Cachegrößen, sowie die Anzahl von möglichen Prozessoren in einem Mehrprozessor-System auf.

Hinsichtlich der einzelnen Prozessoren können folgende interessante Neuerungen verzeichnet werden. Beim SPARC64 VIIIfx wächst die Anzahl der Integer- (Float-) Register von 32 (32) auf 64 (128). Eine weitere interessante Neuheit ist, dass Programme Zugriff auf Cache-Strukturen erhalten. Dabei kann ein Programm Daten in zwei verschiedenen Cachebereichen ablegen. Einer der beiden Bereiche wird für Daten genutzt, die im Allgemeinen nur einmal gelesen werden und somit nur von der *örtlichen Lokalität* profitieren. Der andere Bereich wird für Daten verwendet, die ein hohes Maß an *zeitlicher Lokalität* aufweisen, also oft gelesen und geschrieben werden. Das Speichersystem des Power7-Prozessors wird bis zu 400 GByte pro Sekunde an Bandbreite bereitstellen. Der Intel-Teraflop ist ein Prozessorprototyp, der im Rahmen des Tera-scale-Computing-Programms entwickelt wurde und basiert auf einer Netzwerkarchitektur [84]. Der 2008 vorgestellte Larrabee ist ein Many-Core-InO-Prozessor, bei dem die Kerne über einen Bus miteinander verbunden sind. Die 32 oder mehr Kerne sind Pentium-Kerne, die um Vektorfunktionalität (512 Bit SIMD-Operationsbreite, 32 Register, 8 16 Bit Bitmasken Register) erweitert wurden [134]. Die Probleme der Cache- und TBL-Überlastung sollen beim Larrabee damit Umgangen werden, dass alle Hardware-Threads eines Kerns unter Verwendung gemeinsamer Daten und Code an der gleichen Aufgabe arbeiten.

Name	Kerne	Millionen Transistoren	Cache	Mehr-prozessor
Fujitsu Sparc64 VIIIfx [74]	8	760	5MB L2	n.b.
AMD Magny-Cours [73, 72, 71]	12	900	12x512KB L2 2x6MB L3	4
SUN Rainbow Falls [76]	16x8SMT	n.b.	16xL2	n.b.
IBM Power7 [75, 77]	8x4SMT	1200	32MB L3 8x256KB L2	32
Intel Larrabee [149, 3]	32x8SMT	n.b.	n.b.	n.b.
Intel Terraflop [84, 18]	80	100	n.b.	n.b.
Intel Nehalem [95, 85, 151]	4x2SMT	731	8MB L3	8
Intel Nehalem EX [95, 43]	8x2SMT	2300	24MB L3	8

Tabelle 2.2: Übersicht von Prozessoren und ihren Eigenschaften

2.1.5.4 Das Testsystem und der Intel-Core-i7

Im Weiteren wird die Intel-Nehalem-Architektur am Beispiel der Intel-Core-i7-Serie erläutert, da Messungen und Vergleichstests auf diesem Prozessortyp vorgenommen wurden und Grundkenntnisse der Architektur notwendig sind, um die verschiedenen Schlussfolgerungen und Analysen verstehen bzw. nachvollziehen zu können. Die Nehalem-Mikroarchitektur ist ein „Tock“ in Intels Tick-Tock-Strategie und stellt die neu entwickelte Mikroarchitektur auf Basis der 45nm Strukturgröße[8] dar. Der Core-i7 besteht aus bis zu vier superskalare OoO-Kernen mit einer SMT-Architektur und kann somit bis zu acht Hardware-Threads ausführen. Die Cachehierarchie besteht aus einem *inklusiven* 8 MByte großen L3-Cache (40 Zyklen Latenz, 16-Wege assoziative) und pro Kern aus einem 256 KByte großen L2-Cache (11 Zyklen, 8-Wege) sowie einem L1-Instruktions- und einem L1-Daten-Cache je 32 KByte groß (4 Zyklen, 4-Wege). Das ccNUMA-Speichersystem (Def. 2.9) (QPI) kann maximal 48 GByte Speicher pro Prozessor verwalten und liefert eine maximale Bandbreite von 32 GByte pro Sekunde [112, 99, 97, 30, 173]. Die QPI-Komponente ist zuständig für den Datentransport von Prozessor zu Prozessor und von Prozessor zum I/O-System, dabei stellt QPI ein Socket zu Socket Bus dar und liefert 25 GByte an Bandbreite (s. Abb. 2.15). Die Nehalem-EX-Architektur ist eine Weiterentwicklung der Nehalem-Architektur [95, 43] und erlaubt bis zu acht Kerne pro Prozessor und jeder Prozessor kann maximal 128 GByte Speicher verwalten.

QPI – Quick Path Interconnect

[8] folgende Tick ist Westmere (32nm)

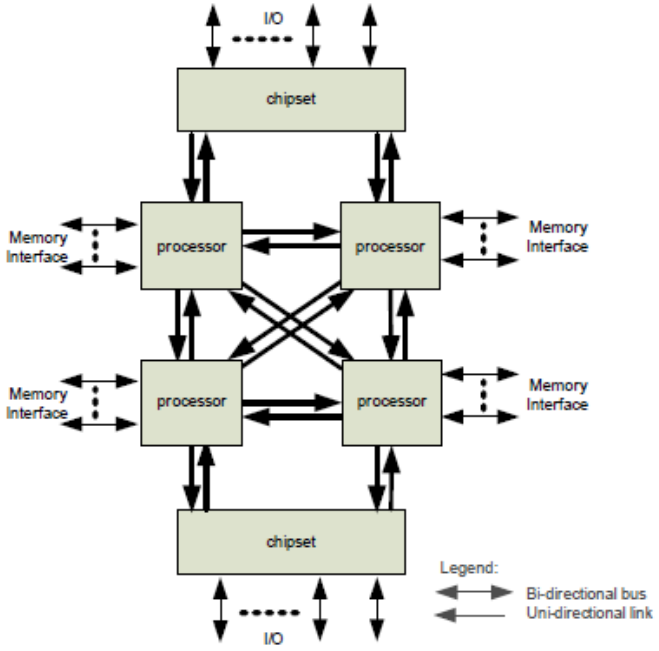


Abbildung 2.15: Intel-Quick Path Interconnect [99]

Bei dem Testsystem handelt es sich um ein Mehrkern-Mehrprozessor-System, bei dem zwei Intel-Core-i7-Prozessoren (E5520) verbaut sind. Die maximale Taktfrequenz liegt bei $2,26 GHz$. Die Cachezeilengröße beträgt 128 Byte, die SIMD-Breite ist 128 Bit. In der ccNUMA Konfiguration kontrolliert jeder Prozessor einen sechs GByte großen Primärspeicher. Als Betriebssystem diente Windows in Version 7 64 Bit Professional.

Fazit: Mehrkern-Prozessoren bieten eine hohe Parallelität auf Prozessorebene. Die Konsequenz für die Programmierung ist ein zunehmender Zwang der Nutzung von Techniken der parallelen Programmierung, da sonst keine Leistungssteigerung für Programme erreicht werden kann. Dazu kommen große SIMD-Einheiten auf den Prozessoren, wie z.B. durch Intel-Advanced-Vector-Extensions [50]. Die Caches auf den Prozessoren werden zunehmend größer und die Software bekommt Einfluss auf die Platzierung und Verwaltung der Daten im Cache. Dazu wird eine steigende Anzahl von heterogenen bzw. homogenen Kernen auf einem einzelnen Prozessor Platz finden.

2.2 Konzepte der parallelen Programmierung

In Abschnitt 2.2 werden Konzepte der parallelen Programmierung vorgestellt. Zunächst wird die Notwendigkeit an paralleler Programmierung von Programmen in Mehrkern-Zeitalter anhand des Amdahlschen Gesetzes [11] und Gustafsons Beobachtung [63] motiviert. Anschließend werden die Begriffe Prozess, Thread und Task definiert. Abschliessend werden Methoden und Vorgehensweisen vorgestellt, um parallele Programme zu

erstellen. Dazu werden verschiedene parallele Programmierumgebungen präsentiert und miteinander verglichen.

Zunächst sollen jedoch einige Begrifflichkeiten geklärt werden. Im Folgenden wird als Ort der Abarbeitung eines Programms der Begriff Hardware-Thread genutzt (Def. 2.15), mit dem Begriff Prozessor wird auf die gesamte physische Komponente, das Package, verwiesen.

Definition 2.16 (Thread) Einen **Ausführungspfad** eines sich in der Abarbeitung befindendes Programm bezeichnen man allgemein als **Thread**. □

Später in Abschnitt 2.2.3 wird auf spezielle Arten und Eigenschaften von Threads eingegangen, sowie die Beziehung zwischen Programm, Prozess und Thread erläutert. Ein paralleles Programm kann dabei zu unterschiedlichen Zeiten eine von eins verschiedene Anzahl von Threads aufweisen.

Definition 2.17 (Nebenläufigkeit) Mit **Nebenläufigkeit** (auch Quasiparallelität genannt) wird auf eine serielle aber nicht weiter definierte Reihenfolge in der Abarbeitung von Threads referenziert. □

Definition 2.18 (Parallelität) Parallelität beschreibt die tatsächliche Gleichzeitigkeit in der Ausführung von Threads. □

Im Kontext von parallelen Programmen wird oft zwischen Nebenläufigkeit und Parallelität unterschieden [108]. Der Unterschied war speziell auf Systemen mit einem einzigen Hardware-Thread wichtig, da dieser zu einem bestimmten Zeitpunkt nur ein einzelnen Thread ausführen kann. Im Zeitalter der Mehrkern-Rechnerarchitekturen ist dieser Unterschied nicht mehr entscheidend und wird somit in dieser Arbeit nicht weiter berücksichtigt und somit wird im Folgenden ausschließlich von Parallelität gesprochen.

2.2.1 Notwendigkeit von Parallelität

Die Auswirkungen der Verschiebung von Einzelkern- zu Mehrkern-Rechnerarchitekturen birgt für die Softwareentwicklung große Herausforderungen. Die Mehrheit der heutigen Programme sind zum großen Teil seriell in ihrer Ausführung und werden den Vorteil der wachsenden Anzahl von Kernen nicht nutzen können [143]. Um das Potential, die Mehrkern-Rechnerarchitekturen bieten, nutzen zu können, werden parallele Programme benötigt. Abbildung 2.16 verdeutlicht die Entwicklung der möglichen Leistungsfähigkeit von seriellen bzw. parallelen Programmen im Zeitalter der Steigerung der Taktraten (GHz-Ära) gegenüber der Mehrkern-Ära [98] und macht deutlich, dass nur die Nutzung von parallelen Programmen ein gutes Leistungsverhalten bieten kann. Eine wichtige Frage dabei ist: Wie hoch muss der Grad an Parallelität eines Programms sein?

2.2.1.1 Bestimmungsmöglichkeiten paralleler Leistung

Um die Frage nach dem benötigten Grad an Parallelität zu beantworten, muss eine Maßzahl für das Leistungsverhalten im Kontext von parallelen Programmen definiert werden.

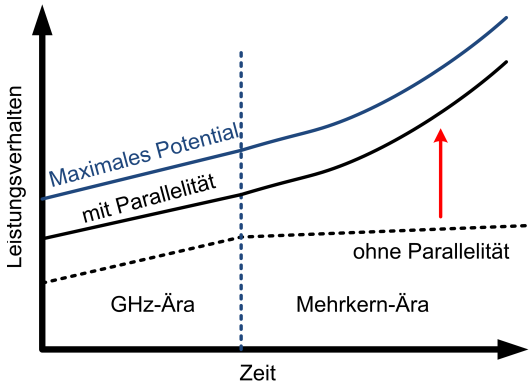


Abbildung 2.16: Notwendigkeit für Parallelität in der Mehrkern-Ära

Bei der Bewertung eines parallelen Programms stellt seine Laufzeit $T_p(n)$ ein wesentliches Kriterium dar. Dabei wird die Laufzeit für eine gegebene Problemgröße n sowie eine maximale Anzahl von zur Verfügung stehenden Hardware-Threads p bestimmt. Um die Leistung des parallelen Programms zu bestimmen, wird die parallele Laufzeit mit der Laufzeit der sequentiellen Ausführung $T_1(n)$ verglichen [79, 108, 142, 143], der somit zur Definition des parallelen *SpeedUps* (vgl. Def. 1.8) führt:

$$S_p(n) = \frac{T_1(n)}{T_p(n)}$$

2.2.1.2 Das Amdahlsche Gesetz

Das Amdahlsche Gesetz untersucht, wie sich der *SpeedUp* eines Programms bei der Erhöhung der Anzahl der Hardware-Threads und konstanter Problemgröße verhält [11, 81]. Dazu wird das Programm in einen seriellen Anteil (f) und seinen parallelen Anteil $(1-f)$ zerlegt, so dass sich für den parallelen Teil einen linear *SpeedUp* ergibt:

$$S_p(n) = \frac{T_1(n)}{fT_1(n) + \frac{(1-f)}{p}T_1(n)} = \frac{1}{f + \frac{(1-f)}{p}}$$

Definition 2.19 ($SpeedUp_{max}$) Der **maximale *SpeedUp*** $SpeedUp_{max}$ für ein gegebenes Programm mit einem nicht parallelisierbaren Anteil von f ist nach dem Amdahlschen Gesetz gegeben mit:

$$SpeedUp_{max} = \lim_{p\to\infty} S_p(n) = \frac{1}{f}$$

□

Abbildung 2.17 zeigt wie sich unterschiedliche Parallelisierungsgrade auf $SpeedUp_{max}$ auswirken. Demnach ist bei einem Parallelisierungsgrad von 90% bereits mit 128 Hardware-Threads ein Maximum erreicht, und eine weitere Erhöhung von Rechenressourcen würde zu keiner weiteren Verbesserung der Laufzeit führen.

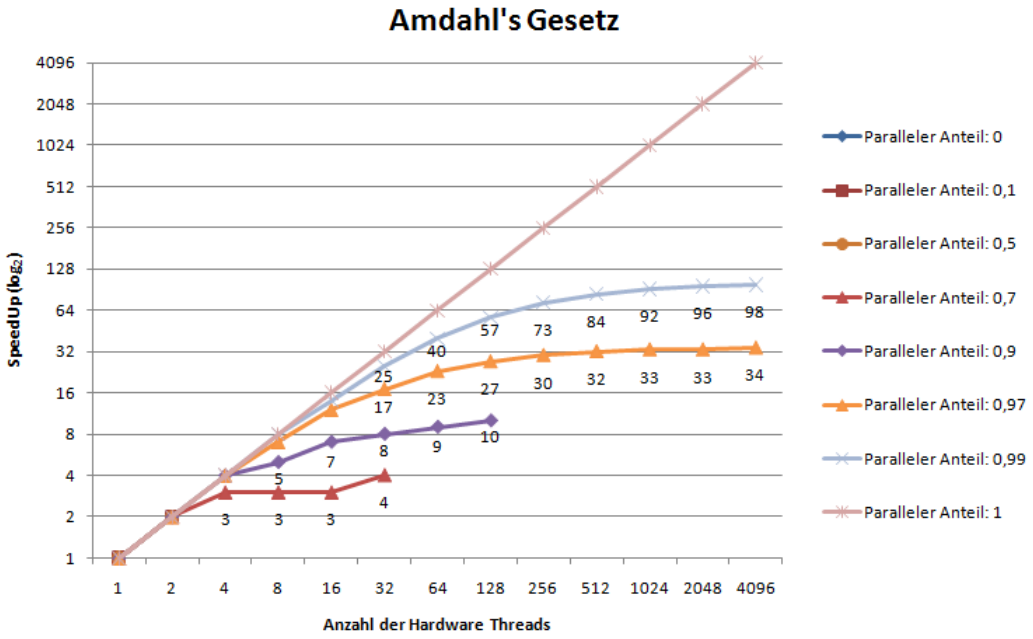


Abbildung 2.17: $SpeedUp_{max}$ für unterschiedliche Grade der Parallelisierung

2.2.1.3 Gustafson's Beobachtung

Die Beobachtung von Gustafson begründet sich darauf, dass im Amdahlschen Gesetz nur die Anzahl der Hardware-Threads verändert wird [63, 108, 143]. Tatsächlich werden aber Probleme komplexer und Daten für die Auswertung größer. Auf Grund dieser Beobachtung erschien es Gustafson bei der Nutzung von Parallelität sinnvoller, auch das Problem mit der Anzahl der Hardware-Threads wachsen zu lassen und definierte den maximalen $SpeedUp_{max}$ wie folgt:

$$SpeedUp_{max} = f + (1 - f)p$$

Fazit: Aus dem Amdahlschen Gesetz und Gustafson's Beobachtung ergeben sich folgende wichtige Erkenntnisse bzgl. des Grades an Parallelität eines Programms:

(i) Der serielle Anteil sollte so gering wie möglich sein.

(ii) Der serielle Anteil sollte bei Vergrößerung des Problems möglichst wenig mitwachsen.

(iii) Die Problemgröße und der serielle Anteil definieren die maximale Anzahl von zu verwendenden Ressourcen.

2.2.2 Techniken der parallelen Programmierung

Im Folgenden wird betrachtet, wie ein paralleles Programm erstellt und ausgeführt werden kann. Techniken zur parallelen Programmierung sind seit vielen Jahren im Einsatz und werden verwendet, um Programme zu beschleunigen, bzw. einzelne Teile eines Programms voneinander zu lösen. Es ist somit kein wirklich neuer Programmierstil notwendig, sondern vielmehr eine Anpassung der Granularität [142]. Nach [142] sind zur Erstellung paralleler Software folgende Grundbegriffe und Kenntnisse hilfreich:

- Wie wird beim Entwurf eines parallelen Programms vorgegangen?
- Welche Eigenschaften der parallelen Hardware sollen zu Grunde gelegt werden?
- Welches parallele Programmiermodell soll genutzt werden?
- Wie kann der Leistungsvorteil eines parallelen Programms gegenüber dem äquivalenten sequentiellen bestimmt werden?
- Welche parallele Programmierumgebung oder -sprache soll genutzt werden?

Die Punkte 2 und 4 wurden bereits in diesem Kapitel behandelt (s. Abschn. 2.1 und Abschn. 2.2.1.1). Die restlichen Fragen sollen in den folgenden Abschnitten diskutiert werden.

2.2.2.1 Entwurf paralleler Programme

Die Grundidee bei der parallelen Programmierung ist es, mehrere Befehlsströme zu erzeugen, die parallel ausgeführt werden können. Die Überführung eines sequentiellen Programms in ein paralleles Programm nennt man **Parallelisierung**.

Für die Parallelisierung existieren viele Paradigmen, die entweder durch neue Sprachen, Spracherweiterungen und oder Bibliotheken implementiert werden. Eines dieser Paradigmen definiert Teilaufgaben (engl. Tasks) und läuft auf Shared-Memory-Architekturen. Das **Taskparadigma** ist sehr gut für Mehrkern-Prozessoren geeignet und basiert auf der Separierung von Tasks und Threads [143].

Definition 2.20 (Task) Eine **Task** stellt eine atomare Einheit bzgl. der Ausführung bzw. des Schedulings dar. □

Bei der Parallelisierung werden die auszuführenden Aufgaben eines Programms in Tasks zerlegt (s. Abb. 2.18). Die Zerlegung muss unter Beachtung von Daten-und Kontrollabhängigkeiten vorgenommen werden. Die Größe einer Task wird als **Granularität** bezeichnet, wobei eine grobe Granularität daraufhin weist, dass zwischen den einzelnen Tasks nur wenige Interaktionen stattfinden. Bei einer feinen Granularität ist meist ein hoher Anteil an Interaktion vorhanden.

Die Ausführung verschiedener Tasks wird durch Threads ermöglicht, denen Hardware-Threads (s. Def. 2.15) zugeordnet werden. Die Zuordnung von einzelner Tasks zu Threads

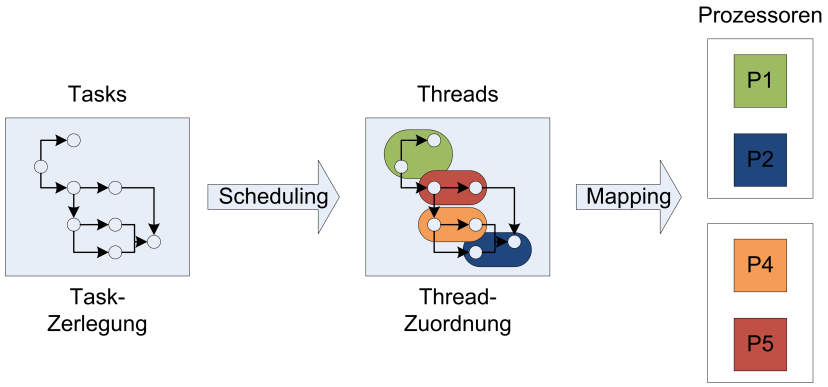


Abbildung 2.18: Abbildung von Tasks auf Threads und Hardware-Threads

wird **Scheduling** genannt, dabei existieren **statische** und **dynamische** Schedulingverfahren. Die Abbildung von Threads auf Hardware-Threads kann entweder explizit vom Programm oder durch das Betriebssystem geschehen [143, 142].

Eine andere Art der Parallelisierung ergibt sich aus der **Datenparallelität**. Dabei wird die gleiche Aufgabe parallel auf unterschiedliche Daten ausgeführt. Die beiden Arten der Parallelisierung schließen sich nicht gegenseitig aus, sondern können in unterschiedlicher Weise miteinander kombiniert werden (näheres in Abschn. 2.2.2.3).

2.2.2.2 Parallele Programmiermodelle

Der Entwurf eines parallelen Programms basiert immer auf einer abstrakten Sicht auf das parallele System, auf dem die Software abgearbeitet werden soll. Diese abstrakte Sicht wird als **paralleles Programmiermodell** bezeichnet und beschreibt ein paralleles Rechnersystem aus der Sicht, die sich dem Softwareentwickler durch Systemsoftware wie das OS, der parallelen Programmiersprache oder Bibliothek, Compiler und Laufzeitbibliothek bietet. Es existieren eine Vielzahl von parallelen Programmiermodellen.

Für eine systematische Herangehensweise geben Rauber et al. folgende Kriterien an [142]:

- Auf welcher Ebene soll das Programm parallelisiert werden: Instruktionsebene, Anweisungsebene, Prozedurebene?
- Wie soll die Parallelität gekennzeichnet werden (implizit oder explizit)?
- In welcher Form sollen die parallelen Programmteile angegeben werden?
- Wie erfolgt die Abarbeitung der parallelen Programmteile im Verhältnis zueinander?
- Wie findet der Informationsaustausch zwischen den parallelen Programmteilen statt: Kommunikation oder gemeinsame Variablen?

- Welche Formen der Synchronisation können genutzt werden?

Ebenen der Parallelität geben an, an welcher Stelle bzw. auf welchen Ebenen unabhängige und damit parallele Aufgaben eines Programms existieren. Man kann Parallelität bzgl. der folgenden Ebenen unterscheiden, wobei durchaus mehrere Ebenen zur Parallelisierung genutzt werden können:

- **Instruktionsebene.** Hierbei handelt es sich um die parallele Ausführung mehrerer Prozessoranweisungen (ILP) (s. Abschn. 2.1.2.2).
- **Anweisungsebene.** Bei der Parallelität auf Anweisungsebene werden mehrere Anweisungen auf dieselben oder verschiedene Daten parallel ausgeführt (SIMD, Def. 2.8).
- Bei der **Schleifenebenen**-Parallelität werden die unterschiedlichen Iterationen über den Schleifenrumpf parallel ausgeführt. Dazu ist es notwendig, dass zwischen den einzelnen Schleifeniterationen keine oder nur bestimmte Datenabhängigkeiten existieren. Wenn die Schleifeniterationen keine Datenabhängigkeiten aufweisen, spricht man von einer **parallelen Schleife**.
- Bei der Parallelität auf **Funktionsebene** werden einzelne Funktionen des Programms parallel ausgeführt. Dabei kann es zu Daten- oder Kontrollabhängigkeiten kommen, die durch geeignete Synchronisationsmechanismen beachtet werden müssen.
- **Programmebene. SPMD** ist ein paralleles Programmiermodell bei dem mehrere Instanzen des gleichen Programms starten, die dann asynchron auf verschiedenen Daten arbeiten.

SPMD – Single Program Multiple Data

Datenabhängigkeit in Schleifen kann zu einer Reduktion der möglichen Parallelität führen, daher geben Chandra et al. [33] für das Datenabhängigkeitsproblem in Schleifen folgenden Lösungsansatz an:

1. Erkennung
2. Klassifizierung
3. Auflösung

Bei der Klassifizierung werden folgende Abhängigkeiten unterschieden: (i) **Flussabhängigkeit**, dabei wird in der Iteration I_n der Wert D_k geschrieben und in der Iteration I_m $(m > n)$ der Wert D_k gelesen. (ii) **Antiabhängigkeit**, wird gekennzeichnet durch ein Lesen von D_k in Iteration I_n und ein Schreiben von D_k in I_m $(m > n)$. (iii) Die **Ausgabeabhängigkeit** wird darüber charakterisiert, dass in den Iterationen I_n und I_m $(m > n)$ der Wert D_k geschrieben wird.

Die Abhängigkeiten (ii) und (iii) lassen sich auflösen, die Flussabhängigkeit dagegen ist eine echte Abhängigkeit, die nicht aufgelöst werden kann. Trotzdem können Schleifen mit Flussabhängigkeiten unter bestimmten Umständen parallelisiert werden [143].

Beispiel 2.7: Schleife mit Flussabhängigkeit

```
int i,a[N];
a[0] = f(0);
for(i=1;i<N;++i)
    a[i] = a[i-1] + f(i);
```

Der Programmausschnitt zeigt eine Schleife mit Flussabhängigkeit. Eine Möglichkeit der Parallelisierung ist, das Ergebnis parallel in zwei Durchläufen zu berechnen.

Zunächst wird das Array a in P Partitionen unterteilt und jeweils für die k-te Partition $(a[k_0], ..., a[k_s])$ das erste Element auf $a[k_0] = f(k_0)$ gesetzt, sowie die restlichen Elemente $(1, ..., s)$ über die obige Schleife berechnet und damit die Summe aller Werte in der jeweiligen Partition gebildet und im letzten Element gespeichert $S_k = a[k_s] = \sum_{i=0}^{s} f(i)$. Diese Berechnung kann für alle P Partitionen parallel ausgeführt werden, da nun keine Abhängigkeiten mehr unter den Partitionen bestehen.

Im zweiten Schritt werden dann für jede Partition die eigentlichen Ergebnisse $a[i]$ berechnet, wobei die zuvor berechneten Teilergebnisse S_j genutzt werden. Dazu wird zu jedem Listelement der i-ten Partition die Summe alle S_j $(j < i)$ hinzu addiert. Auch der zweite Schritt kann auf alle P Partitionen parallel ausgeführt werden, solange die entsprechenden Vorgängerpartitionen bereits Schritt eins beendet haben.

Insgesamt werden bei diesem Verfahren zwar etwas mehr als doppelt so viele Additionsoperationen durchgeführt, allerdings lässt bei genügend großem N und P guter *SpeedUp* erzielen.

Abschnitt 2.3.3.1 wird nochmals die Parallelisierung auf Schleifenebene behandeln und für beide Schleifenarten ein Modell zur Berechnung des *SpeedUp* bzw. des Parallelitätsgrads vorstellen.

Explizite oder implizite Parallelität bestimmt, ob und wie die Parallelitäten des Programms gekennzeichnet werden. Bei der *impliziten Parallelität* werden keine Angaben zur Parallelisierung gemacht. Beispiele dafür sind parallelisierende Compiler und funktionale Programmiersprachen. Die *explizite Parallelität* kann in zwei Gruppen unterteilt werden: (i) Explizite Parallelität mit **impliziter Zerlegungsangabe**. Dabei werden Teile mit Parallelisierungspotenzial explizit gekennzeichnet, z.B. einer parallelen Schleife. Um die Parallelisierung selbst muss sich der Programmierer aber nicht kümmern. (ii) Bei der expliziten Parallelität mit **expliziter Angabe** ist der Programmierer auch für die Zerlegung des Programmteils in parallele Tasks zuständig. Dabei wird unterschieden, ob auch eine explizite Zuordnung an Threads und Hardware-Threads oder eine explizite Kommunikation formuliert wird. Bei Programmiermodellen mit expliziter Kommunikation und Synchronisation muss der Programmierer alle Details der parallelen Abarbeitung angeben.

Bei der **Form zur Angabe paralleler Programmteile** existieren viele verschiedene Möglichkeiten, mehr dazu in Abschnitt 2.2.4. Die **Abarbeitung der einzelnen parallelen Programmteile** unterscheidet man in **synchron** und **asynchron**. Eine Parallelisierung durch SIMD-Verarbeitung ist immer synchron. Bei der asynchronen Abarbeitung

kann von Zeit zu Zeit eine explizite Synchronisation notwendig werden.

Für den **Informationsaustausch** stehen verschiedene Möglichkeiten zur Verfügung. Zum einen können bei einem gemeinsamen Adressraum der beteiligten Threads gemeinsame Variablen verwendet werden, die dann von allen beteiligten Thread gelesen und geschrieben werden. Zum anderen können andere Arten der Kommunikation verwendet werden, z.B. durch das versenden von Nachrichten. Bei den **Formen der Synchronisation** unterscheidet man zwischen **Barriere-Synchronisation** und dem **gegenseitigen Ausschluss** durch Sperrmechanismen und bedingtes Warten. Bei der Barriere-Synchronisation müssen alle beteiligen Threads an der Barriere warten, bis auch der Letzte diesen Punkt der Ausführung im Programm erreicht hat. Mit dem gegenseitigen Ausschluss soll verhindert werden, dass zwei oder mehr Threads gleichzeitig innerhalb bestimmter Abschnitte der Ausführung stehen.

2.2.2.3 Muster der parallelen Programmierung

Die Möglichkeiten der Interaktion der einzelnen Threads eines parallelen Programms bzw. die Koordinationsstruktur der beteiligten Threads werden Muster der parallelen Programmierung vorgestellt. Die Erzeugung der beteiligten Threads kann entweder statisch zu Beginn der Ausführung geschehen oder dynamisch während der Laufzeit. Je nach Art des Threads und Betriebssystem kann die Erzeugung unterschiedlich aufwendig sein. Folgende Muster der parallelen Programmierung existieren [142, 143]:

- **Fork-Join.** Beim Fork-Join-Muster erzeugt (Fork) der ausgeführte Thread dynamisch einen oder mehrere neue Threads, die danach parallel abgearbeitet werden. Zu einem späteren Zeitpunkt wartet der erzeugende Thread, bis die erzeugten Threads beendet wurden (Join) und setzt dann erst die Ausführung fort.

- Das **Parbegin-Parend**-Muster definiert einen Programmbereich, innerhalb dessen alle Anweisungen parallel abgearbeitet werden können. Dazu werden mehrere Threads erzeugt, die die jeweiligen Anweisungen des Blocks ausführen. Erst nach Beendigung aller Threads wird das Programm weiter durch einen Thread ausgeführt.

- Beim **Master-Slave**- oder **Master-Worker**-Muster kontrolliert ein einzelner Thread (Master) die Abarbeitung des Programms. Er erzeugt die Worker- oder Slave-Threads, die die eigentliche parallele Programmausführung erledigen. Die Aufgabenverteilung geschieht durch den Master oder auch durch die Worker selbst.

- Das **Client-Server**-Modell ähnelt dem MPMD-Modell. Es stammt aus dem verteilten Rechnen, wo mehrere Client-Rechner mit einem als Server dienenden Mainframe verbunden sind. Parallelität auf Serverseite wird durch parallele Auswertung der Client-Anfragen ermöglicht. Bezüglich Shared-Memory und einzelner Rechner kann dieses Modell auf Threads erweitert werden. Dabei stellt ein einzelner Thread oder auch mehrere Threads den Server dar. Die Client-Threads können an die Server-Threads-Anfragen senden. Die Antworten werden von den Server-Threads berechnet und an die Client-Threads zurückgesendet.

MPMD – Multiple Program Multiple Data

- **Pipelining** ist eine besondere Form der Zusammenarbeit, bei der Daten zwischen den Threads weitergereicht werden. Eine bestimmte Anzahl von Threads $T_1, ..., T_n$ sind in einer logischen Reihenfolge so angeordnet, dass die Ausgabe von T_i als Eingabe von T_{i+1} dient. Alle n Threads können somit parallel arbeiten. Pipelining stellt eine spezielle Form der funktionalen Zerlegung dar.

- **Taskpooling** ist ein weiteres wichtiges Muster im Umfeld von Mehrkern-Rechnerarchitekturen. Der **Taskpool** ist dabei eine spezielle Datenstruktur, in der noch zu bearbeitende Tasks abgelegt werden. Bei der parallelen Abarbeitung der Tasks entnimmt ein Thread eine Task aus dem Pool und bearbeitet sie. Während der Abarbeitung kann er neue Tasks erzeugen und in den Taskpool einfügen. Der Vorteil ist, dass eine feste Anzahl von Threads verwendet werden kann und der Aufwand der Erzeugung einer Task sehr viel geringer ist, als für die Erzeugung eines Threads. Die Anzahl der Threads kann somit unabhängig von der Problemgröße definiert werden. Durch das dynamische generieren von Tasks können adaptive und irreguläre Anwendungen sehr viel effizienter abgearbeitet werden.

- Als letztes soll das **Produzenten-Konsumenten**-Muster betrachtet werden. Bei diesem Muster stehen Daten im Mittelpunkt. Ein Thread führt immer eine bestimmte ihm zugeordnete Aufgabe aus, dazu greift er auf einen gemeinsamen Datenpuffer zu. Es existieren Threads, die Daten in den Puffer hineinstellen (Produzenten) und andere, die Daten aus dem Puffer entnehmen (Konsumenten).

2.2.3 Programme, Prozesse und Threads

Dieser Abschnitt soll den Begriff des Threads genauer definieren. Dabei wird es um die Einordnung in Bezug auf die Begriffe Programm und Prozess gehen, sowie die verschiedenen Konzepte der Abarbeitung von Threads.

Die Begriffe *Prozess* und *Thread* werden bzgl. ihrer Abbildung auf die Hardware in den verschiedenen Betriebssystemen unterschiedlich verwendet. Im Folgenden soll das Prozessmodell des *Windows NT* Betriebssystems als Grundlage für die weitere Betrachtung genutzt werden [119, 138]. Dabei stellt ein Prozess die Abbildung des virtuellen Speichers des Programms auf Teile des realen Speichers dar. Threads werden bei der Ausführung auf die einzelnen Hardware-Threads abgebildet, und das Betriebssystem verwaltet die einzelnen Komponenten wie Speicher, CPU, Ein- und Ausgabegeräte und stellt sie dem Nutzer über Schnittstellen zur Verfügung. Es ist somit eine Art Virtualisier der Hardware [137]. Können Betriebssysteme mehrere Programme nebenläufig ausführen, spricht man von **Multi-Tasking** Fähigkeit, im Fall, dass der Computer mehrere Hardware-Threads besitzt, auch von **Multi-Processing**.

Definition 2.21 (Prozess) Ein **Prozess** ist ein sich in der Ausführung befindliches Programm und hat exklusives Recht auf bestimmte ihm zugeteilte Ressourcen wie Speicher und Dateien. Der einem Prozess zugeordnete Speicher ist ein virtueller Speicher mit linearem Adressraum und wird durch das Betriebssystem auf einzelne Bereiche bzw. Seiten des Primär- und Sekundärspeichers umgesetzt. □

Ein Prozess kann einen oder mehrere Threads (Def. 2.16) gleichzeitig haben. Alle Threads des gleichen Prozesses können auf die gemeinsamen Ressourcen zugreifen, insbesondere arbeiten alle Threads eines Prozesses auf dem gleichen virtuellen Speicher. Prozesse sind in den meisten Fällen unabhängig, während Threads immer nur innerhalb eines Prozesses existieren und besitzen außer einen eigenen Stack, Programmzähler sowie einen threadlokalen Bereich für globale Daten (TLS) [166, 142] keinerlei eigene Ressourcen. Dabei wird bzgl. der Parallelität eines Programms in Inter- und Intraprozess-Parallelität unterschieden.

TLS – Thread Local Storage

Definition 2.22 (Interprozess-Parallelität) Interprozess-Parallelität kennzeichnet sich dadurch aus, dass ein Programm über mehrere Prozesse parallelisiert wird. Jeder Prozess arbeitet in seinem eigenen virtuellen Speicher. Damit können die verschiedenen Prozesse nicht direkt auf gemeinsame Daten im Speicher zugreifen. Die Kommunikation wird über Schnittstellen des Betriebssystems ermöglicht. □

Definition 2.23 (Intraprozess-Parallelität) Ein Programme mit **Intraprozess**-Parallelität wird durch einen einzigen Prozess dargestellt, der mehrere Threads für die parallele Ausführung besitzt. □

Die Intraprozess-Parallelität auf Programmebene ist sehr effizient, da die Kommunikation auch ohne das Betriebssystem erfolgen kann. Zugriffe auf gemeinsame Daten müssen allerdings synchronisiert werden (s. Abschn. 2.2.3.2). Im Rest der Arbeit wird nur noch auf die Intraprozess-Parallelität eingegangen.

2.2.3.1 Ausführungsmöglichkeiten von Threads

Ein Thread kann einen der folgenden fünf verschiedene Zustände einnehmen und zwischen ihnen wechseln (s. Abbildung 2.19): (i) Neu erzeugt, (ii) lauffähig, (iii) laufend, (iv) wartend und (v) beendet.

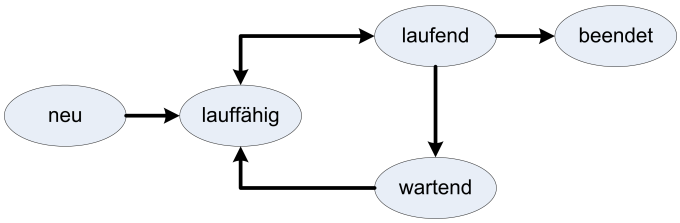


Abbildung 2.19: Zustände eines Threads

Bei der Ausführung durch das OS wird jedem Thread ein virtueller Hardware-Thread als Ausführungseinheit zugeordnet. Da ein Thread diese Ressource exklusiv nutzt, muss das Betriebssystem jedem virtuellen Hardware-Thread einen physischen Hardware-Thread zuordnen. Dabei kann es vorkommen, dass nicht genügend physische Hardware-Threads vorhanden sind, in diesem Fall muss das OS dafür sorgen, dass eine Möglichkeit existiert, um mehreren virtuellen Hardware-Threads nacheinander einen physischen

Hardware-Thread zuzuordnen. Das Zuordnen eines physischen Hardware-Threads zu einem virtuellen Hardware-Thread wird als **Scheduling** bezeichnet. Die Neuzuordnung eines physischen Hardware-Threads zu einen anderen virtuellen Hardware-Thread wird **Kontextwechsel** genannt. Der Kontextwechsel zwischen zwei Threads unterschiedlicher Prozesse (**Prozess-Kontextwechsel**) ist sehr viel aufwendiger als der Kontextwechsel zwischen zwei Threads des gleichen Prozesses (**Thread-Kontextwechsel**). Im Folgenden soll das **Zeitscheibenverfahren**, auch **Time-Sharing** genannt, als eine mögliche Schedulingstrategie betrachtet werden.

Beim Time-Sharing teilen sich mehrere virtuelle Hardware-Threads einen physischen Hardware-Thread [143]. Jeder virtueller Hardware-Thread kann somit für eine gewisse Zeitspanne den physischen Hardware-Thread nutzen. Die Zeitspanne kann dabei entweder durch das OS oder den Thread selbst festgelegt werden.

Definition 2.24 (Kooperatives Multithreading) Beim **kooperative Multithreading** wird die Zeitspanne der Benutzung des physischen Hardware-Threads von dem Thread selbst festgelegt. Der Thread selbst gibt seine Nutzungsrechte an das OS zurück, erst dann erfolgt der Kontextwechsel. □

Definition 2.25 (Präventives Multithreading) Dagegen wird die Zeitspanne beim **präventiven Multithreading** durch das OS festlegt. Hat der Thread seine Zeit verbraucht, wird er durch das OS unterbrochen und es erfolgt ein Kontextwechsel. □

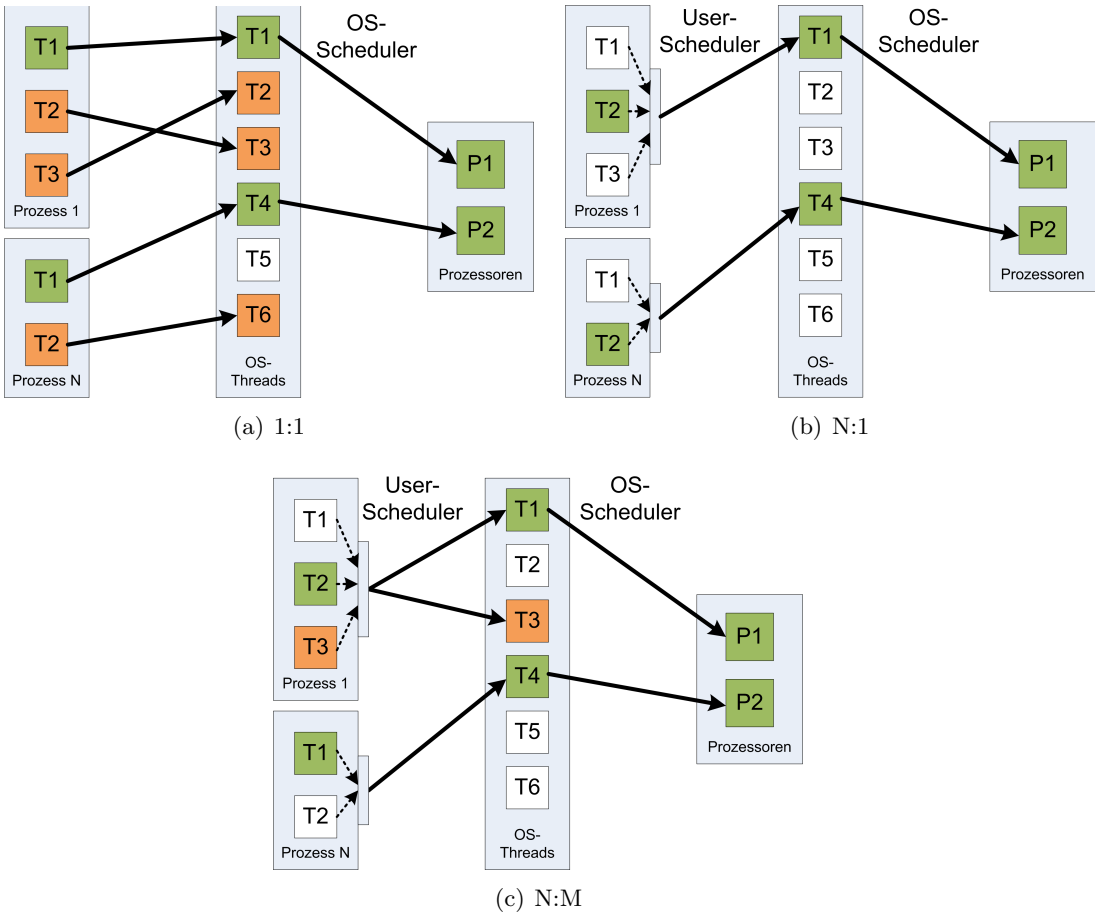
Mit präventivem Multitasking können ausgeglichenere, faire Zeitpläne erzeugt werden, da der Kontextwechsel durch das Betriebssystem initiiert wird. Aus diesem Grund wird in den meisten Betriebssystemen ein präventiver Multithreading-Scheduler verwendet [143, 166]. Ein Nachteil des kooperativen Multithreading ist, dass es leicht zu **Deadlock**-Situationen kommen kann. Präventives Multithreading zwischen mehreren Threads eines Programms kann aber auch zu Problemen führen, z.B. das sogenannte „Cache-Cooling“. Beim Cache-Cooling wurden die Cache lokalen Daten des Threads nach einem Kontextwechsel verdrängt, und es kommt zu einer Erhöhung von Cache-Misses (Def. 2.3). Ein anderes Problem hängt damit zusammen, dass das OS keinerlei Informationen darüber hat, was das Programm bzw. die einzelnen Threads gerade machen. Das Betriebssystem kann die Abarbeitung eines Threads an fast jeder Stelle unterbrechen und damit Effekte namens „Convoying“ und „Priority Inversion“ hervorrufen.

Definition 2.26 (Convoying) Beim **Convoying** erlangt ein Thread T_i eine Sperre S und wird unterbrochen ehe er S wieder befreien kann. Anschließend wird ein Thread T_k ausgeführt, der ebenfalls die Sperre S erlangen möchte, diese aber nicht erlangen kann. Da die Sperre bereits durch T_i besetzt ist, kann T_k entweder seine Zeit mit Warten auf S verbringen oder die Kontrolle an das OS zurückgeben. In beiden Fällen wird T_k vergeblich ausgeführt und durch die Kontextwechsel unnötiger Weise Zeit verschwendet. □

Definition 2.27 (Prioritätsumkehrung) **Prioritätsumkehrung** (engl. Priority Inversion) beschreibt die Situation, in der ein niedrig priorisierter Thread T_L eine Sperre S erhalten hat, unterbrochen wird und in Folge ein höher priorisierter Thread T_H S

sperren möchte. T_H kann S nicht erlangen, so dass er in den Zustand „wartend“ übergehen muss. Falls nun noch ein weiterer Thread T_M mit einer höheren Priorität als T_L lauffähig ist, wird T_L nicht ausgeführt, so dass T_H weiterhin blockiert bleibt. □

Auf Grund des Convoying-Problems und der Prioritätsumkehrung ist präventives Multithreading auf Programmebene oft nicht erwünscht. Um trotz eines präventiven Multithreading-Schedulers kooperatives Multithreading nutzen zu können, wurden Benutzerthreads eingeführt und Threads zwischen **Betriebssystemthreads** (Kernel-, OS-Threads) und **Benutzerthreads** (User-Threads) unterschieden. OS-Threads werden durch das OS verwaltet und jeder Prozess hat mindestens einen OS-Thread. User-Threads werden ohne das OS verwaltet.



(a) 1:1 (b) N:1

(c) N:M

Abbildung 2.20: Abbildungsmöglichkeiten User-Thread – OS-Thread

Bei der Abbildung von User-Threads auf OS-Threads werden drei Möglichkeiten (s. Abb. 2.20) unterschieden:

- **1:1** Jedem User-Thread wird genau ein Kernel-Thread zugeordnet (s. Abb. 2.20(a)).

- **N:1** N User-Threads werden auf einen Kernel-Thread abgebildet. Das Betriebssystem hat hierbei keine Informationen bzw. Kontrolle über die User-Threads. Das N:1 Modell ist sehr leicht zu implementieren. Es kann aber die Vorteile von mehreren Hardware-Threads nicht nutzen, da immer nur ein User-Thread ausgeführt werden kann (s. Abb. 2.20(b)).
- **N:M** Beim N:M Modell werden N User-Threads auf M Kernel-Threads abgebildet (s. Abb. 2.20(c)).

2.2.3.2 Parallele Datenzugriffe im gemeinsamen Adressraum

Wie bereits zuvor beschrieben, ist bei der Intraprozess-Parallelität (Def. 2.23) der gemeinsame Adressraum von großem Vorteil. Er macht aber auch die Verwendung von Synchronisation notwendig, um die Korrektheit im Bezug auf die parallele Ausführung sicherzustellen. Die Synchronisation kann dabei implizit oder explizit erfolgen. Eine implizite Synchronisation ist dabei immer vorzuziehen, da die explizite Synchronisation mit zusätzlichem Aufwand verbunden ist und meist problem- und fehleranfälliger ist [143, 142]. Ein wichtiges Konzept im Umfeld der Programmierung von intraprozessparallelen Programmen ist die **Thread-Sicherheit**.

Definition 2.28 (Thread-Sicherheit) Thread-Sicherheit (engl. Thread Safety) in Bezug auf einen definierten Teil eines Programms bedeutet, dass eine beliebige Anzahl von Threads in beliebiger Reihenfolge gleichzeitig (parallel) diesen Programmteil durchlaufen kann und immer ein korrektes Ergebnis liefert [143, 108]. □

Thread-Sicherheit kann auf mehrere Arten erzeugt werden, einige davon sind:

- **Re-entrancy** ist eine Eigenschaft von Programmfunktionen. Sie ist definiert durch: (i) Die Funktion verwendet keinerlei globale[9] oder statische Variablen. (ii) Das Ergebnis hängt nur von den Funktionsparametern ab. (iii) Die Funktion ruft ausschließlich re-entrante Funktionen auf.
- **Atomare Operationen** sind Operationen, bei denen Daten in einer nicht unterbrechbaren Operation **komplett** verändert werden. Dazu werden spezielle Prozessorbefehle verwendet, bei denen eine bestimmte Anzahl von Bytes durch einen Befehl verändert werden. Darunter existieren z.B. Befehle wie Addieren, Inkrementieren und Austauschen. Meist sind diese Befehle auf Basisdatentypen beschränkt und können nicht auf selbst definierte Strukturen angewendet werden. Atomare Operationen bilden die Grundlage für viele Thread-Sperrmechanismen.

TLS – Thread Local Storage

- **TLS** ist ein für jeden Thread exklusiver Bereich, um private Kopien von globalen Variablen zu halten. Das Konzept des TLS ermöglicht es, dem Programmierer mit dem gleichen Variablennamen auf unterschiedliche Speicherbereiche zu referenzieren und damit Thread-Sicher zu programmieren.

[9] Prozess global

- **Transaktionaler Speicher** (TM) kann verwendet werden, um Lese- und Schreibvorgänge von unterschiedlichen Threads zu synchronisieren. Dabei unterscheidet man zwischen Software- und Hardware-TM [79, 10, 78, 163].

TM – Transactional Memory

- **Mutual Exclusion,** der gegenseitige Ausschluss, ist ein weiteres Konzept, um auf gemeinsame Daten zuzugreifen. Dieses Konzept erlaubt die Serialisierung von Zugriffen beim gleichzeitigen Lesen und Schreiben einer Variablen. Zu Problemen kommt es speziell dann, wenn ein Programmteil auf zwei oder mehr Teile von gemeinsamen Daten gleichzeitig zugreifen muss. Unter diese Probleme fallen „Datenwettläufe“ (engl. Data Races), „zeitkritische Abläufe“ (engl. Race Conditions), Deadlocks, Livelocks sowie das „Verhungern“ (engl. Starvation) [166]. Der gegenseitige Ausschluss ist aber auch sehr leistungsanfällig auf Grund möglicher Zugriffskonflikte (engl. Contention). Bei einer hohen Nutzung kann es schnell zu einer vollständigen Serialisierung der beteiligen Threads kommen, so dass keinerlei Parallelität mehr existiert.

2.2.3.3 Mechanismen des gegenseitigen Ausschlusses

Für den gegenseitigen Ausschluss existieren verschiedene Mechanismen und Umsetzungen darunter das Konzept des **Mutex** (von Mutual Exclusion), des **Semaphores** und der **Bedingungssynchronisation**. Das Konzept des Mutex funktioniert wie folgt: Bevor die Ausführung in einen kritischen Bereich hineingeht, muss das Mutex erlangt werden, falls es durch einen anderen Thread blockiert wird, muss „gewartet“ werden. Sobald das Mutex im Besitz des jeweiligen Threads ist, kann er mit der Ausführung fortfahren und am Ende des kritischen Bereichs das Mutex wieder freigeben. Unter Umständen müssen noch andere Threads über die Freigabe informiert werden. Eine Evaluierungen von verschiedenen Mechanismen des gegenseitigen Ausschlusses bzgl. ihres Einflusses auf die Leistung wird in [102, 36] gegeben. Die einzelnen Konzepte können anhand der folgenden Punkte unterschieden werden:

(i) Mit Hilfe welcher Datentypen wird die Sperren realisiert?

(ii) Wie wird auf eine blockierende Sperre gewartet?

(iii) Welche Operationen werden nachdem Entsperren notwendig?

(iv) Wird der Sperrmechanismus auf Kernel- oder Userebene umgesetzt?

Sperren können unterschiedlichen Typs sein, z.B. Bool'scher Datentyp für die Realisierung des Mutex, Integer für die Realisierung eines Semaphores bis hinzu komplexen Datentypen wie Events-, Datei- oder Threadhandels (Bedingungssynchronisation).

Beim Warten auf die Sperre können drei mögliche Alternativen unterschieden werden, wie auf die Situation vom Thread reagiert werden kann. Zum einen besteht die Möglichkeit des sogenannten **Spinnings** [92, 26]. Dabei bleibt der Thread im Zustand des Laufens und prüft in einer Schleife, ob sich die Synchronisationsvariable geändert

hat und letzten Endes frei geworden ist. Spinning hat zwei Nachteile: (i) Die Zeit für Warten auf die Freigabe wird komplett verwendet und kann nicht von anderen Threads[10] genutzt werden. (ii) Da die Synchronisationsvariable immer wieder gelesen werden muss, kann es durch Cache-Kohärenz zu enormen Belastungen für die beteiligten Prozessoren und Prozessorverbindungen kommen und damit zu Leistungsabfällen.

Eine weitere Möglichkeit für das Warten besteht darin, dass bei einer bereits vergebenen Sperre der Thread seine Kontrolle freiwillig an das OS abgibt. Er bleibt allerdings im Zustand *lauffähig*; der Scheduler kann somit den Thread bei nächster Gelegenheit wieder ausführen. Der Thread muss dann erneut die Sperre prüfen und kann gegebenenfalls in seiner Ausführung fortfahren oder wieder die Kontrolle abgeben. Der Vorteil dieses Ansatzes ist, dass die Zeit für das Warten sinnvoll durch andere Threads genutzt werden kann. Ein wesentlicher Nachteil besteht in den teuren Kontextwechseln und einer von der Sperre unabhängigen die Wiederausführung durch den Scheduler.

Die letzte Möglichkeit ist eine Erweiterung der zweiten, in sofern, dass der Thread nicht im Zustand *lauffähig* bleibt, sondern in den Zustand *wartend* übergeht. Die Wiederausführung durch den Scheduler kann jetzt in Abhängigkeit von der eigentlichen Sperre erfolgen. Dies hat zur Folge, dass der Thread nicht vergeblich ausgeführt wird und faire Warteschlangen implementiert werden können.

Bei der Entsperrung liegt der Unterschied hauptsächlich in der Frage, ob mit der Freigabe der Sperre auch eine Benachrichtigung an einen bzw. alle wartenden Threads erfolgt (Bedingungssynchronisation).

User- und Kernelebene definiert, an welcher Stelle der Sperrmechanismus umgesetzt wird. So kann eine Sperre auf Userebene nicht auf beliebige Signale warten, sondern meist nur einen Kontextwechsel verursachen, oder durch Spinning die Zeit überbrücken. Dagegen kann ein Sperrmechanismus auf Kernelebene auf spezielle Signale warten[11]. Nachteil von Sperren auf Kernelebene ist ihr großer Aufwand bei der Ausführung. Da der Kernel im Ring 0 läuft, User-Code aber in Ringen höherer Stufe, muss das Programm eine Änderung am Privilegsmodus vornehmen, welche aufwendig ist. Dazu kommen wie z.B. im NT-Betriebssystem ein bzw. zwei (hin und zurück) Thread-Kontextwechsel, da Kernelfunktionen durch einen anderen Thread ausgeführt werden [137].

Fazit: Zusammenfassend können folgende Punkte als Schlüssel für effiziente und skalierbare Intraprozess-Parallelisierung von Programmen aufgeführt werden [33]: (i) Anteil des parallelen Programmteils, (ii) Granularität, (iii) Last-Balancierung , (iv) Lokalität und (v) Synchronisation. Neben den bereits aufgeführten Problemen, wie False-Sharing (Def. 2.5), Prioritätsumkehrung (Def. 2.27), Convoying (Def. 2.26) und Zugriffe auf gemeinsame Variablen (s. Abschn. 2.2.3.2 Mutual Exclusion) sind noch folgende Punkte, als Quellen von Problemen und starken Leistungseinbrüchen zu nennen:

- **Speicherkonsistenz,** die im englischen als „Memory Consistency“ bezeichnete Eigenschaft von Prozessoren, bezieht sich auf das Verhalten der Prozessoren bzgl.

[10] außer bei SMT
[11] Abhängig vom OS

Speicheroperationen [79]. Dabei wird zwischen starker und schwacher Konsistenz unterschieden. Bei der starken Konsistenz wie sie die Intel IA-32- und Intel64-ISAs implementieren, werden die Speicheroperationen letzten Endes in der vom Programm definierten Reihenfolge ausgeführt. Bei Prozessoren mit schwacher Konsistenz, wie z.B. dem Intel-Itanium, dem IBM-Power-PC und Alpha und dem Sun-SPARC, ist das nicht der Fall. Die Prozessoren können die Schreibvorgänge fast beliebig vertauschen, was in einem seriellen Programm keine weiteren Probleme verursacht, bei parallelen Programmen kann es allerdings zu Problemen führen. Dazu existieren auf diesen Architekturen sogenannte „Fence“ Operationen, die den Prozessor veranlassen alle geänderten Werte hinaus zuschreiben. Diese Fence Operationen sind auch implizit in allen atomaren Lese- und Schreiboperationen enthalten. Zu beachten ist dabei, dass die Operationen sehr teuer sind, also bis zu mehrere 100 Zyklen brauchen.

- **Speicherallokation** stellt eine besondere Herausforderung für die Intraprozess-Parallelität dar, da nur ein Heap für alle Threads eines Prozesses existiert, müssen die Allokation und Deallokation synchronisiert werden. Das kann aber bei hoher Frequentierung zu nicht skalierbaren und damit langsamen Programmen führen. Hier ist auch nochmal auf das Problem mit ccNUMA verwiesen, welches ebenfalls bei falscher Teilung zu Leistungsverlusten führen kann. Es ist somit notwendig, auch die Speicherallokation (-deallokation) effizient zu gestalten.

- **Überbuchung** (engl. Oversubscription) ist das Problem, dass deutlich mehr Software-Threads als Hardware-Threads vorhanden sind und damit eine hohe Anzahl von Kontextwechseln mit Problemen wie dem Cache-Cooling entsteht.

2.2.4 Parallele Programmierumgebungen

In diesem Abschnitt werden verschiedene Programmierumgebungen vorgestellt, die eine Entwicklung parallele Programme ermöglichen. Dabei wird die Intraprozess-Parallelität (Def. 2.23) im Vordergrund stehen. Für die Entwicklung paralleler Programme stehen verschiedene Möglichkeiten zur Verfügung. Neben funktionalen Programmiersprachen wie Haskell [69] oder F# [121] existieren Erweiterungen für Java, C, C++ und andere bekannte Programmiersprachen. Dazu kommen neue Sprachen, wie z.B. OpenCL [106], Cilk [109] und Ct [55, 56, 57]. Im Folgenden sollen vier Möglichkeiten vorgestellt werden, die auf C bzw. C++ aufbauen. Dabei wird auf Ergebnisse der Diplomarbeit von Herrn Nagel zurückgegriffen [127].

2.2.4.1 Thread-Programmierung

Die Thread-Programmierung ist ein bekanntes Programmierkonzept [142, 143]. Thread-Programmierung wird in [143] als *“assembly language of parallel programming”*, also der Assemblersprache für parallele Programmierung, bezeichnet. Der Programmierer erhält zwar ein Maximum an Freiheiten, muss dafür aber mit hohen Aufwand für die Programmierung, die Fehlersuche und die Wartung rechnen. Es obliegt dem Programmierer, sich

um alle Einzelheiten (Zerlegung, Zuordnung von Task an Threads, Synchronisation) zu kümmern und explizit auszudrücken. Nach Reinders [143] neigt Thread-Programmierung oft zu wenigen Threads mit vielen Sperren und skaliert nicht bzw. nicht gut.

2.2.4.2 OpenMP

OpenMP ist ein Standard, der von verschiedenen Firmen und Organisationen für Shared-Memory-Architekturen formuliert wurde, um parallele Programme zu entwickeln [33, 143, 98, 20, 142]. Der Standard ist für die Programmiersprachen C, C++ sowie Fortran definiert. OpenMP basiert auf expliziter Parallelität mit impliziter Zerlegung und arbeitet nach dem Fork-Join-Modell. Der Programmierer muss sich zwischen einem statischen, dynamischen und unterstützendem (engl. guided) Schedulingverfahren entscheiden [143].

Beispiel 2.8: OpenMP For-Schleife mit Reduktion

```
int create_sum(int * list, int count)
{
   int i,sum = 0;

#pragma omp parallel for \
       shared(list,count) \
       private(i) \
       reduction(+: sum)
   for(i=0;i<count;++i)
      sum += list[i];

   return sum;
}
```

Beispiel 2.8 zeigt den Quelltext für die Berechnung der Summe über eine Liste von Zahlen mit Hilfe von OpenMP.

OpenMP besteht aus Compileranweisungen, die sogenannten Pragmas, eine Menge von Laufzeitfunktionen und Umgebungsvariablen. Die Pragmas stellen dabei keine Sprachkonstrukte der Programmiersprache dar, vielmehr kann der Programmierer mit ihrer Hilfe dem Compiler mitteilen, was mit den nachfolgenden Anweisungen geschehen soll. OpenMP Pragmas in C haben den Präfix *#pragma omp*. Über die Pragmas können unter anderen Geltungsbereiche für Variablen, Parallele-Schleifen und -Blöcke, kritische Bereiche, Barrieren und atomare Operationen gekennzeichnet werden. Variablen können als „private“ (also pro Thread eine Kopie) oder „shared“ (eine gemeinsame Variable für alle Threads) markiert werden. Dazu kommt noch „reduction“, was bedeutet, dass jeder Thread eine Kopie erhält auf der er arbeitet, jedoch am Ende die einzelnen Kopien wieder zusammengefügt werden, z.B. durch Addition. Zu diesem Pragmas kommen Methoden für die explizite Synchronisation und die Beeinflussung der Laufzeitumgebung, z.B. Anzahl der Threads. Bezüglich der Funktionalität existieren in OpenMP verschiedene Restriktionen, darunter:

- Die Berechnung der Anzahl der Schleifendurchläufe muss unabhängig von der Schleife sein.
- Die Anzahl der Schleifeniterationen (engl. Trip count), muss berechenbar sein aus Ober- und Untergrenze und des „Stride“, dem Wert um den die Schleifenvariable erhöht wird.
- Schleifen können nicht vorzeitig abgebrochen werden.
- Die Reduktion kann nur auf Built-In-Typen mit einer begrenzten Menge an Operationen ausgeführt werden.

2.2.4.3 Das TBB-Framework

Die Intel Thread Building Blocks (TBBs) sind ein Framework (Def. 1.10), dass speziell für die Parallelisierung von Programmen auf Mehrkern-Prozessoren entworfen wurde [143, 98]. Die TBBs sind für C++ erhältlich und basieren auf Konzepten wie Template- und Generic-Programming [143, 9]. Dabei stellen die TBBs eine Menge von Funktionen, Klassen und Templates bereit, um Programme zu parallelisieren.

Im Unterschied zu OpenMP, bei dem viele Operationen (z.B. Reduktion) nur auf Built-In-Typen definiert sind, können mit den TBBs durch die Verwendung der generischen Programmierung jeglicher Art von Objekten genutzt werden.

Das zentrale Konzept für die TBBs stellt die atomare Ausführungseinheit (engl. Task) (s. Def. 2.20) dar. Einzelne Tasks können definiert und parallel ausgeführt werden, ohne eine Verteilung der Tasks auf die vorhandenen Hardware-Thread anzugeben oder die zeitliche Ausführung zu planen. Die einzelnen Tasks sollten dazu möglichst auf implizite Synchronisation zurückgreifen.

Tasks werden von dem in den TBBs enthaltenen **Taskscheduler** auf die einzelnen Threads, respektive Kerne verteilt. Der Taskscheduler ist auf eine hochperformante Ausführung von Task optimiert und berücksichtigt unter anderen auch Cache-Effekte. Dabei wird jedem Hardware-Thread ein einzelner Thread zugeordnet (1:1 Modell). Die Idee ist, nicht preemptiv zu sein; ein angefangener Task muss beendet werden, bevor der ausführende Thread eine neue Task abarbeitet. Ein Problem entsteht dabei z.B. durch I/O-Operationen. Diese Operationen brauchen meist längere Zeit und blockieren währenddessen den Thread in seiner Ausführung.

Bei Taskausführung wird das **Split-Join**-Muster verwendet. Dabei kann eine ausgeführte Task neue Tasks erzeugen (Split) und kann dann auf die Fertigstellung der erzeugten Tasks warten (Join). Die erzeugten Tasks werden in einem Taskgraphen organisiert.

Definition 2.29 (Taskgraph) Ein **Taskgraph** ist ein DAG, der die Beziehungen zwischen den einzelnen Tasks widerspiegelt. Dabei sind die Knoten die einzelnen Tasks, und die Kanten zeigen auf die Kinder, auf die der Vaterknoten warten muss. □

Für das Split-Join-Muster existieren zwei Möglichkeiten. Bei der ersten und einfacheren Methode muss der Elterntask vor seiner Beendigung auf das Ende aller seiner

Kindertasks warten. Während des Wartens kann er natürlich auch selbst andere Tasks, inklusive seiner eigenen Kinder ausführen. Dabei blockiert der Task solange bis alle Kinder beendet sind; daher wird diese Methode auch **Blocking Style** genannt. Diese Methode ist einfacher zu programmieren, kann aber zu Leistungsproblemen führen. Da jeder Elterntask auf seine Kinder warten muss, kann der Threadstack nicht abgeräumt werden. Das wiederum könnte zu einem nicht begrenzten Wachsen des Stacks führen. Das Problem wird in den TBBs in der Art verhindert, dass der Scheduler niemals Task auf diesem Thread ausführt, die eine geringere Tiefe haben als der tiefste blockierte Task.

Bei der zweiten Methode für das Split-Join-Muster, definiert die Elterntask explizit eine Task für die Fortsetzung, nachdem die Kinder beendet sind; daher auch **Continuation Passing** genannt. Der Vorteil ist, dass der Thread-Stack von der eigentlichen Task gelöst wird und der Scheduler mehr Freiheit in der Wahl der nächsten Task hat.

Im Gegensatz zu OpenMP muss der Programmierer sich nicht für eine **Schedulingstrategie** entscheiden. Stattdessen verwendet der TBB-Scheduler einen **Work-Stealing**-Ansatz [79, 143]. Im Vergleich zu OpenMP stellt dieses Vorgehen eine Mischform zwischen *guided* und *dynamisch* dar, ohne jedoch in die Probleme der Verwaltung einer globalen Task-Warteschlange zu geraten.

Der Taskscheduler nutzt den Taskgraphen, um die nächste Task für einen freien Thread auszuwählen. Dabei stehen grundsätzlich alle vorhandenen Blätter für eine Ausführung zur Verfügung. Die effiziente Auswahl der nächsten Task kann durch Breiten- oder Tiefensuche unter den zur Ausführung bereitstehenden Tasks organisiert werden. Die Tiefensuche ist für die serielle Ausführung ideal, da sie den benötigten Platz minimiert und Cache-Effekte ausnutzt. Bei der parallelen Ausführung dagegen ist die Breitensuche günstiger, da man mit ihr einen deutlich höheren Parallelisierungsgrad erreicht. Die TBBs verwenden einen Ansatz, der das Beste beider Welten darstellt. Diese Strategie wird als „Breadth-first theft and depth-first work“ bezeichnet. Dabei wird die nächste Task zuerst mit Hilfe der Tiefensuche bestimmt. Wird keine Task gefunden, wird eine Task von einem anderen Thread mit Hilfe der Breitensuche „gestohlen“.

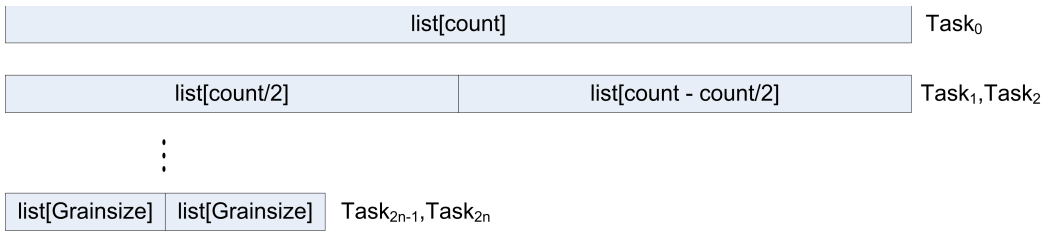


Abbildung 2.21: Zerlegung der For-Schleife durch die TBBs

Mit Hilfe von Abbildung 2.21 soll kurz die Vorgehensweise bei der Parallelisierung der Schleife durch die TBBs dargestellt werden. Am Anfang wird eine Task $Task_0$ erzeugt, die die komplette Berechnung der Liste[12] *list* darstellt. Danach wird *list* rekursive geteilt. In diesem Fall n mal in die Tasks $Task_1, Task_2, ..., Task_{2n-1}, Task_{2n}$. Wenn ein zweiter

[12] Werte in eckigen Klammern geben die Länge an

Thread frei wird, kann er eine Task aus diesem Taskpool stehlen und wird vielleicht $Task_2$ auswählen und diese dann rekursive teilen und bearbeiten. Ein dritter Thread hätte die Wahl zwischen $Task_4$ des ersten Threads oder der entsprechenden Task vom zweiten Thread.

Beispiel 2.9: TBB For-Schleife mit Reduktion

```
1  int create_sum(int * list, int count)
2  {
3     int sum = parallel_reduce(
4        blocked_range<int*>( list, list+count ),
5        0,
6        [](const blocked_range<int*>& r, int init)->int {
7           for( int* i=r.begin(); i!=r.end(); ++i )
8              init += *i;
9           return init;
10       },
11       []( int x, int y )->int {
12          return x+y;
13       }
14    );
15    return sum;
16 }
```

Der oben stehende Quelltext ist eine Implementierung der *FOR*-Schleife mit Reduktion unter Verwendung der TBBs (analog zu Bsp. 2.8). Dabei werden zwei Lambda-Funktionen verwendet [117, 167]. Die erste der beiden Lambda-Funktionen (Zeile 6-10) stellt die Funktion der Schleife bzgl. einer Partition von *list* dar. Die zweite Lambda-Funktion (Zeile 11-13) führt die Reduktion der Ergebnisse zweier Partitionen durch.

Der Vorgang des Teilens geht immer solange, bis nur noch eine bestimmte Anzahl von Elementen zu verarbeiten sind. Die Anzahl wird **Grain-size** genannt. Die Bestimmung der *Grain-Size* ist extrem wichtig für die Leistungsfähigkeit der Parallelisierung. Eine zu klein gewählte *Grain-Size* bedeutet z.B. zu viel Schedulingaufwand. Eine zu groß gewählte *Grain-Size* beschränkt den Grad der Parallelität und führt somit nicht zu einem optimalen Leistungsverhalten.

Am Beispiel 2.9 kann man auch die *implizite* Synchronisation bei der Reduzierung erkennen. $Task_0$ kann erst ausgeführt werden bzw. beendet werden, wenn $Task_1$ und $Task_2$ beendet sind. Es werden somit keinerlei Sperren benötigt, um auf die Ergebnisse zurückzugreifen, die Tasks selbst definieren die Synchronisation. $Task_0$ wird automatisch dann ausgeführt, wenn alle Kinder im Taskgraphen beendet sind, nicht vorher. Das gilt natürlich auch entsprechend für alle rekursiv erzeugten Tasks.

2.2.4.4 Das ConcRT-Framework

Microsoft Concurrency Runtime (ConcRT) ist ein Framework (Def. 1.10) zur Erstellung paralleler Programme [120, 138, 122] und besteht aus vier Komponenten: Einem

Ressourcen-Manager, dem ihm zugeordneten Scheduler und zwei dem Scheduler zugeordneten Bibliotheken. Zum einen eine Asynchrone-Agenten-Bibliothek, die Komponenten seines Programms durch ein asynchrones Kommunikationsmodell verbinden kann und auf der Modellierung des Datenflusses basiert. Zum anderen eine Bibliothek für parallele Muster (PPL). Sie stellt verschiedene Konzepte für die parallele Programmierung zur Verfügung.

PPL – Parallel Pattern Library

Das ConcRT-Framework und das TBB-Framework basieren auf den gleichen Konzepten und gleichen sich stark. Ein Hauptunterschied stellt aber der Scheduler dar. Im ConcRT-Framework werden eigene User-Threads und ein benutzerdefiniertes Scheduling, kurz UMS genannt, implementiert.

UMS – User Mode Scheduling

Mit UMS können Threads verwaltet werden, ohne den OS-Scheduler zu nutzen. Dazu wird auf das 1:1 Thread-Modell zurückgegriffen und jedem Hardware-Thread ein UMS Scheduler mit einer beliebige Menge von User-Threads zu geordnet. Nach Probert [138] hat Microsoft es dabei geschafft den Kontextwechsel, der bei Neuzuordnung eines User-Thread zum Kernel-Thread notwendig ist, solange heraus zu zögern, bis der aktive User-Thread in den Kernel wechselt. Dadurch sind die Kontextwechsel zwischen den User-Threads deutlich weniger aufwendig.

Ein weitere Unterschied ist, dass bei einer Blockierung eines Kernel-Threads (z.B. durch Sperren oder I/O-Operationen) der entsprechende User-Thread in den Zustand *wartend* versetzt wird und die Kontrolle wieder zurück an den entsprechenden UMS-Scheduler gegeben wird, der einen anderen User-Thread ausführen kann.

Auf Grund der starken Ähnlichkeit zum TBB-Framework wird an dieser Stelle auf ein entsprechendes Beispiel für die For-Schleife mit Reduktion verzichtet.

2.2.4.5 Vergleich der Programmierumgebungen

Im Folgenden werden die vier zuvor vorgestellten Programmierumgebungen zum Erstellen von parallelen Programmen zusammen mit zwei weitere Möglichkeiten, der (i) automatischen Vektorisierung und der (ii) automatischen Parallelisierung, verglichen. Zunächst werden die Vor- und Nachteile der einzelnen Methoden zur Erstellung paralleler Programme vorgestellt. Anschließend werden die Programmierumgebungen OpenMP und das TBB-Framework bzgl. ihres Leistungsverhaltens verglichen. Für diesen Zweck greift die vorliegende Arbeit auf Teile der Diplomarbeit von Herrn Nagel zurück [127].

Definition 2.30 (Automatische Vektorisierung) Die automatische Vektorisierung ist eine Technik, die während der Programmübersetzung vom Compiler verwendet werden kann, um die Leistungsfähigkeit des Programms zu erhöhen. Dabei werden einzelne aufeinanderfolgende sequentielle Befehle zu SIMD-Befehlen (Def. 2.8) überführt, so dass eine Datenparallelität (s. Abschn. 2.2.2.1) entsteht [98]. □

Definition 2.31 (Automatische Parallelisierung) Automatische Parallelisierung ist die Technik, bei der der Compiler abhängigkeitsfreie Schleifen (s. Seite 51) parallelisiert [98]. □

Methode	Vorteile	Nachteile
Automatische Vektorisierung	• Automatische Verwendung von ILP durch den Compiler. • Kann mit anderen Parallelisierungstechniken kombiniert werden.	• Der erzeugte Bytecode ist Prozessor abhängig und läuft damit nicht auf jedem Prozessor (selbst bei gleicher ISA). • Spezielle Compilerunterstützung ist notwendig.
Automatische Parallelisierung	• Der Compiler erstellt automatisch Multithreaded Code. • Kann mit anderen Parallelisierungstechniken kombiniert werden.	• Funktioniert nur auf Schleifen, die der Compiler als unabhängig erkennt. • Spezielle Compilerunterstützung ist notwendig.
Thread-Programmierung	• Maximale Kontrolle und Flexibilität. • Keinerlei Unterstützung durch den Compiler benötigt.	• Komplexer Code, schwer wartbar und zeitraubend bei der Fehlersuche. • Das gesamte Management und alle Synchronisation der Threads liegt bei dem Programmierer.
OpenMP	• Möglichkeit von großen Leistungsgewinnen mit geringem Aufwand. • Erlaubt die inkrementelle explizite Parallelisierung durch Compilerdirektiven. • Pragmas ermöglichen gleiche Codebasis für die parallele als serielle Version.	• Nur wenig Nutzerkontrolle bzgl. der Threads und ihrer Eigenschaften. • Schleifenparallelisierung nur für Basisdatentypen.
Intel TBB, Microsoft ConcRT	• Kein spezialisierter Compiler notwendig durch die Verwendung von Standard C++ Code. • Das Framework übernimmt die Threadgenerierung, -verwaltung und das Scheduling.	• Nur für die Verwendung von C++ Programmen geeignet.

Tabelle 2.3: Vergleich der Möglichkeiten zum Erstellen paralleler Programme

Die Tabelle 2.3 führt die unterschiedlichen Vor- und Nachteile der einzelnen Methoden auf. Dabei wurden das TBB-Framework und das ConcRT-Framework innerhalb eines gemeinsamen Punktes beschrieben. Der Grund liegt darin, dass die TBBs ab Version 3.0 mit ConcRT kompatible sind und sich beide Frameworks somit nur marginal unterscheiden. In der Diplomarbeit von Herrn Nagel wurden daher nur OpenMP mit den Intel-TBBs verglichen. Dabei erwiesen sich die beiden untersuchten Ansätze als gleich ausdrucksstark und geeignet zur Entwicklung paralleler Programme. Das TBB-Framework mit seiner Vielzahl an gut dokumentierten Templates bot jedoch einen höheren Komfort bei der Implementierung von Algorithmen. Hinsichtlich der Leistung wurden keine nennenswerten Unterschiede beobachtet. Bei der Evaluierung auf verschiedenen Testsystemen boten beide Frameworks einen nahezu optimalen *SpeedUp*. Beide Frameworks zeigten in der Arbeit ein großes Skalierungspotential bei wachsender Datenmenge und entsprechender verfügbarer Hardware. Durch die Erkenntnisse und Erfahrung aus der Arbeit von Herrn Nagel wurde für die prototypische Implementierung in der vorliegenden Arbeit (s. Abschn. 1.3) das TBB-Framework als parallele Programmierumgebung gewählt.

2.3 Das Hardwaremodell und seine Operatoren

Wie im Abschnitt 2.1 diskutiert wurde, sind heutige Prozessoren enorm komplex und heterogen. Um in diesem Umfeld ein RDBMS zu programmieren, das alle Möglichkeiten zur Leistungssteigerung der verschiedenen Prozessoren und Systeme ausnutzt, ist bisher mit einem enormen Aufwand verbunden. Einerseits ist es notwendig, jede einzelne Architektur sehr gut zu kennen; andererseits wird der Quelltext für das RDBMS extrem unleserlich, da viele Sonderfälle eingearbeitet werden müssen.

Im letzten Teil des zweiten Kapitels wird daher ein Hardwaremodell, sowie seine Operatoren vorgestellt, die den Prozess der Entwicklung von RDBMSen auf unterschiedlichen Mehrkern-Rechnerarchitekturen vereinfachen soll. Auf Abbildung 2.22 ist der Teil des Frameworks markiert, mit dem sich dieser Abschnitt beschäftigen wird.

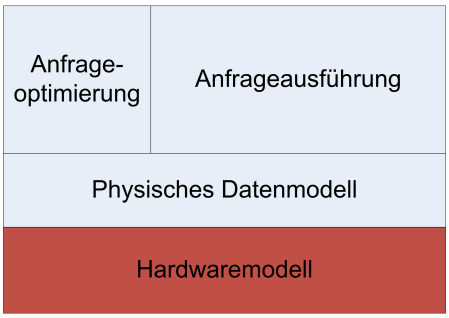
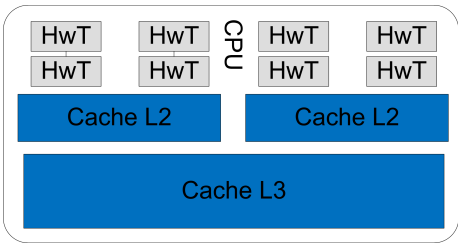


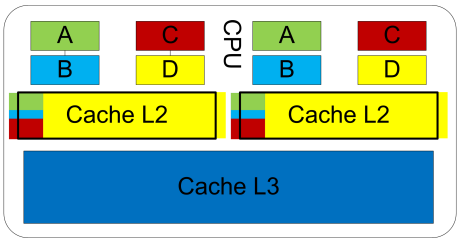
Abbildung 2.22: Framework: Hardwaremodell

Anhand von zwei Beispielen soll noch einmal das Problem verdeutlicht werden, dabei wird das erste Beispiel 2.10 die Idee hinter dem Hardwaremodell geben. Das zweite

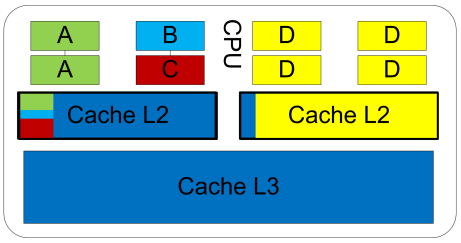
Beispiel 2.11 verdeutlicht das Problem von Hardware abhängigen Befehlen und die Einschränkung möglicher Algorithmen und ihrer Implementierungen.



(a) Beispiel Hardwarearchitektur



(b) Naive Aufteilung



(c) Optimale Aufteilung

Abbildung 2.23: Zusammenspiel von Programm und Rechnerarchitektur

Beispiel 2.10: Zusammenspiel von Programm und Rechnerarchitektur

Gegeben sei ein Mehrkern-Rechner mit acht Hardware-Threads (Def. 2.15) und einer dreistufigen Cachehierarchie (Def. 2.1), bei der sich je vier Hardware-Threads einen 512 KByte großen L2-Cache und alle zusammen sich einen 12 MByte großen L3-Cache teilen (s. Abb. 2.23(a)).

Ein paralleles Programm – bestehend das aus den Blöcken A, B, C und D – soll auf die einzelnen Ressourcen verteilt werden. Die Blöcke A, B und C arbeiten in einer Pipeline, wogegen D unabhängig läuft. Sie benötigen für die Verarbeitung der einzelnen Daten einen bestimmten Anteil der seriellen Zeit t und eine bestimmte Menge an Speicher A:[$\frac{t}{4}$,20 KByte], B:[$\frac{t}{8}$,10 KByte], C:[$\frac{t}{8}$,30 KByte] und D:[$\frac{t}{2}$,500 KByte], wobei D für jedes Datum auf die gleichen 500 KByte zugreift (z.B. in Form einer Hashtabelle). Bei der Aufteilung kann eine naive (s. Abb. 2.23(b)) oder eine optimale Anordnung (s. Abb. 2.23(c)) vorgenommen werden.

Die naive Aufteilung hat folgende Probleme: (i) Die Datenmengen von A, B, C und D können aus Platzmangel nicht alle gleichzeitig im jeweiligen L2-Cache gehalten werden. (ii) Die Nichtbeachtung des gemeinsamen Datenzugriffs von D verursacht zusätzlichen Aufwand durch Cache-Kohärenz der beiden L2-Caches. (iii) Da die zeitlichen Abhängigkeiten der einzelnen Bereiche nicht berücksichtigt wurden, werden die Hardware-Threads für die Bereiche B und C nur zur Hälfte ausgelastet. Die Zeitunterschiede bei der Bearbeitung zwischen der Pipeline $A - B - C$ und D sorgen dafür, dass die Pipeline nach der Hälfte der Zeit fertig ist, während D noch rechnen muss. Der maximale *SpeedUp*

wäre somit zwei. Diese zeitliche Verschiebung sorgt zu einer asynchronen Abarbeitung der einzelnen Daten, was zu zusätzlichen Aufwand durch Speicherladeoperationen führen kann.

Mit der optimalen Aufteilung dagegen können die Probleme (i), (ii) und (iii) direkt vermieden werden. Durch die Anordnung der Bereiche A, B und C oberhalb eines einzelnen L2-Caches wird dessen Größe nicht komplett ausgenutzt. Mit der Verdoppelung des Bereichs A kann die Pipeline in voller Geschwindigkeit laufen, ohne das B und C unnötig warten. Die vierfache Replizierung des Bereichs D und seine Anordnung oberhalb eines einzelnen L2-Caches verhindert zusätzliche Leistungsverluste durch Cache-Kohärenz und sorgt dafür, dass die Pipeline und D im gleichen Tempo arbeiten, es ist somit ein *SpeedUp* von acht möglich, also viermal so viel wie im naiven Fall.

Abstraktion ist ein wichtiges Hilfsmittel, um sich auf die wichtigen Punkte zu konzentrieren, und unnötige bzw. überflüssige Details auszublenden. Die Idee hinter dem Hardwaremodell und seinen Operatoren ist es, zwei Welten voneinander zu trennen. Anstatt einen Experten für die Relationale- und die Hardware-Welt zu haben, hat man zwei Experten. Einer kümmert sich um den Hardwareteil; er implementiert und optimiert die Operatoren für die verschiedenen Systeme und der andere kann mit Hilfe dieser Operatoren seine Algorithmen und Realisierungen von Konzepten der Relationalen-Welt beschreiben.

Beispiel 2.11: Nutzung spezieller Befehle

Im folgenden Beispiel soll aus eine lange Sequenz von Nullen und Einsen (Bit-String Def. 3.11), die Anzahl von Einsen bestimmt werden. Für diese Aufgabe existieren verschiedene Algorithmen zur Berechnung [14]. Ebenfalls existiert auf einigen Mikroarchitekturen ein spezieller Befehl namens *popcnt* für die Art von Berechnung, welcher deutlich schneller ist als generelle Softwarelösungen. Dieser Befehl ist jedoch weder Teil der ISA-IA-32 noch der Intel64-ISA, sondern ist in bestimmten Erweiterungen (SSE4) definiert. Soll das Programm unabhängig von der Rechnerarchitektur sein, muss entweder auf *popcnt* verzichten werden, oder im Quelltext werden zwei Abarbeitungsmöglichkeiten eingebaut. Bei der Abarbeitung muss das Programm dann jedes mal von neuem entscheiden, welchen Ausführungspfad es wählt, was wiederum zu zusätzlicher Arbeit führt (s. Abschn. 2.1.2.2 Pipelineflush Seite 29). Oder es wird auf die Unabhängigkeit von der Rechnerarchitektur verzichtet und das Programm (z.B. über Compileranweisungen) speziell angepasst, was eine große Menge von binären Versionen des Programms zur Folge hat, die gepflegt werden müssen.

SSE4 – Streaming SIMD Extensions 4

2.3.1 Das Hardwaremodell

Das **Hardwaremodell** soll es ermöglichen, das System und seine notwendigen Eigenschaften auf eine einfache Art und Weise zu beschreiben. Dabei stehen folgende Komponenten im Vordergrund: Prozessoren, Caches und Hauptspeicher.

Manegold führt in [114] ein vereinheitlichtes Hardwaremodell zur Beschreibung von Einzel-Prozessorsystemen ein. Er beschreibt den Speicher als eine kaskadierende Hierarchie von N Cachestufen, wobei jede Stufe die Eigenschaften Name C, Größe $|C|$, LineSize $\#C$, Anzahl der Cachezeilen $|\#C|$ und die Art der Assoziativität besitzt (Def. 2.2). Im Folgenden wird dieses Modell erweitert, um es auf Mehrkern-Rechnerarchitekturen anwenden zu können. Dabei wird (i) auf die Darstellung mehrerer Hardware-Threads, Kerne, Prozessoren und ihrer Eigenschaften, (ii) die Teilung von Caches über verschiedene Hardware-Threads, sowie (iii) ccNUMA Konstellationen eingegangen.

In dieser Arbeit wird das vereinheitlichte Hardwaremodell mit Hilfe eines ungerichteten Graphen dargestellt. Die Knoten des Graphen repräsentieren einzelne Prozessoren, Caches und den Hauptspeicher bzw. die Hauptspeicher im Fall von ccNUMA (Def. 2.9). Dazu kommen sogenannte Kommunikationsknoten, die keinerlei Eigenschaften haben, aber verschiedene Teile miteinander verbinden. Die Kanten des Graphen stellen Verbindungen zwischen den einzelnen Teilen dar, auch sie haben Eigenschaften (z.B. Bandbreite).

2.3.1.1 Der Hardware-Graph

Der Hardware-Graph (HW-Graph) stellt alle Ressourcen des Systems, bzgl. Prozessoren, Caches und Hauptspeicher in einem ungerichteten beschrifteten Graphen dar.

Definition 2.32 (HW-Graph) Ein **Hardware-Graph** (HW-Graph) G ist ein Tupel $G = (N, E, L)$, mit $E \subseteq (N \times N)$ und $N \subseteq (\mathcal{P} \cup \mathcal{C} \cup \mathcal{M} \cup \mathcal{K})$. $\mathcal{P}$ ist die Menge aller Hardware-Threads, $\mathcal{C}$ die Menge aller Caches, $\mathcal{M}$ die Menge der Hauptspeicher, $\mathcal{K}$ die Menge von Kommunikationsknoten und L die Menge der Labels ist. □

Beispiel 2.12: Beispiele für HW-Graphen

Abbildung 2.24 zeigt drei Beispiele für einen HW-Graphen. Der Graph auf Abb. 2.24(a) zeigt ein System mit einem Mehrkern-Prozessor. Der Prozessor besitzt vier Kerne, wobei jeder Kern einen einzelnen L1-Cache hat. Je zwei Kerne teilen sich den L2-Cache, sind über einen gemeinsamen Systembus mit dem Hauptspeicher (HS) verbunden und durch den Kommunikationsknoten (K) dargestellt sind.

Abbildung 2.24(b) stellt ein fiktives Mehrkern-System dar, bei dem auch die einzelnen Prozessoren auf dem Chip miteinander kommunizieren können, wie z.B. beim Intel-Larrabee (s. Abschn. 2.1.5.4).

Der Graph auf Abb. 2.24(c) zeigt den HW-Graphen des Testsystems (s. Abschn. 2.1.5.4). Darauf sind die beiden ccNUMA-Speicher zusehen, sowie die beiden Mehrkern-Prozessoren, mit jeweils vier physischen Kernen. Jeder Kern kann mit Hilfe von SMT zwei Hardware-Threads ausführen. Es ist somit ein System von 16 Hardware-Threads.

Da der Kontext dieser Arbeit Hauptspeicherdatenbanken sind, fließt in dieses Modell keinerlei Informationen über die Sekundärspeicher ein. Das Modell ist aber leicht durch einen zusätzlichen Knotentyp erweiterbar. Hierbei muss nur beachtet werden, dass zum

Beispiel Flashspeicherplatten unterschiedliche Eigenschaften bzgl. Lesen und Schreiben haben und damit je nach Modellierung aus dem ungerichteten Graphen ein gerichteter Graph werden muss.

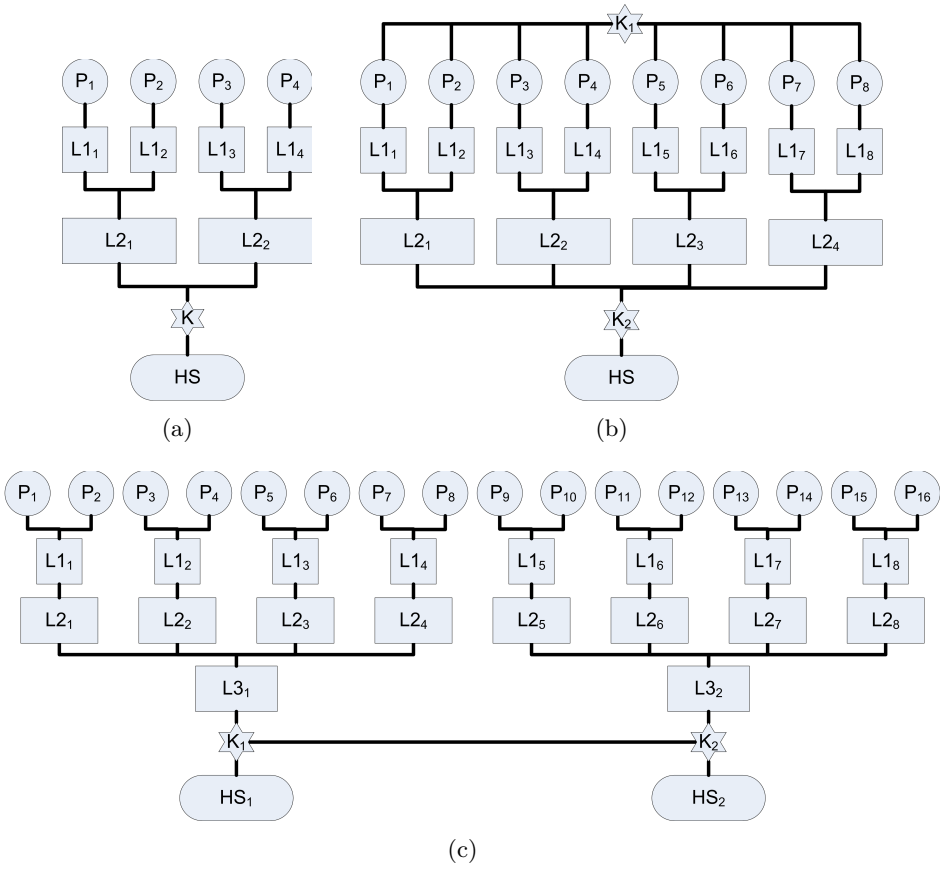


Abbildung 2.24: Beispiele für HW-Graphen

Definition 2.33 (PropEdge) Die Funktion $PropEdge(G, e)$ gibt für den HW-Graphen $G = (N, E, L)$ und eine Kante $e \in E$ die Eigenschaften der Kante e zurück. □

HW-Graphen werden in den Abschnitten 4.3 und 5.3 bei der Anfragebearbeitung genutzt, um einzelnen Operationen des QEPs bestimmte Hardwareressourcen zuzuordnen, was einer **Belegung** entspricht.

Definition 2.34 (Belegung) $A(G) = (G_A, U_N, U_E, L)$ ist eine Belegung eines HW-Graphen $G = (N, E, L)$ wenn: (i) $G_A = (N_A, E_A, L)$ ein zusammenhängender Teilgraph von G mit $N_A \subseteq N$ und $E_A \subseteq E$ ist. Und (ii) die Funktion U_N (U_E) definiert ist, die jedem Knoten (jeder Kante) in G_A einen Wert zwischen 0 und 1 zuordnet. Dieser Wert drückt aus, wie stark die Belegung bzgl. der jeweiligen Ressource ist. Im Folgenden werden wir für die Menge der Knoten (Kanten) der Belegung A kurz $A(N)$ ($A(E)$) verwenden. □

Definition 2.35 (usageEdge) Die Funktion $usageEdge(A, G, e)$ für die Belegung A auf G für die Kante $e \in A(E)$ ist definiert als der Wert von U_E bzgl. e.

$$usageEdge(A, G, e) = U_E(e)$$

□

Definition 2.36 (partOfEdge) Bezüglich einer Menge von Belegungen $\mathcal{A} = \{A_1, ..., A_n\}$ auf G und der Kante e ist das Ergebnis der Funktion $partOfEdge$ die Teilmenge Belegung aus $\mathcal{A}$ bei denen jede Belegung die Kante e in $A(E)$ ist.

$$partOfEdge(\mathcal{A}, G, e) = \{A_i | e \in A_i(E)\} \subseteq \mathcal{A}$$

□

Definition 2.37 (usageSetEdge) Die Funktion $usageSetEdge(\mathcal{A}, G, e)$ bzgl. einer Menge von Belegungen $\mathcal{A}$ auf G und der Kante e ist definiert als

$$usageSetEdge(\mathcal{A}, G, e) = 1 - \sum_{A_i \in partOfEdge(\mathcal{A},G,e)} usage(A_i, G, e)$$

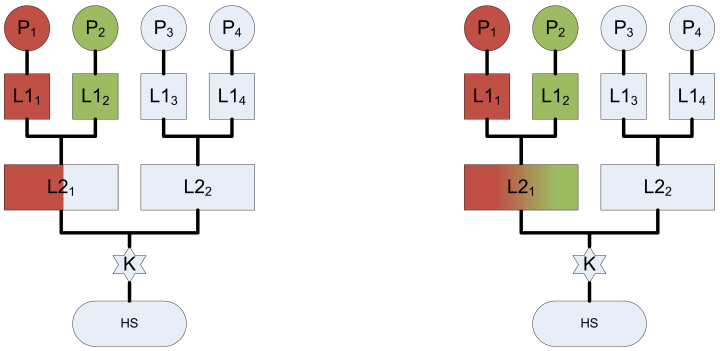
□

Analog seien für Knoten die Funktion $PropNode$, $usageNode$, $partOfNode$ und $usageSetNode$ definiert. Als letztes soll eine Aussage über die Gültigkeit von Belegungen möglich sein.

Definition 2.38 (Valide Belegungen) Die Menge von Belegungen $\mathcal{A}$ auf dem HW-Graphen $G = (N, E, L)$ ist genau dann **valide**, wenn gilt:

$$valid(\mathcal{A}, G) = \begin{cases} true & \text{if } ((\forall e \in E | 0 \leq usageSetEdge(\mathcal{A}, G, e) \leq 1) \bigwedge \\ & (\forall n \in N | 0 \leq usageSetNode(\mathcal{A}, G, n) \leq 1)) \\ false & \text{sonst.} \end{cases}$$

□



(a) Valide (b) Nicht valide

Abbildung 2.25: Belegungsmöglichkeiten des HW-Graphen

Beispiel 2.13: Beispiele Belegungen

Abbildung 2.25(a) zeigt eine valide Belegungsmenge für den HW-Graphen, dabei werden die Hardware-Threads P_1 und P_2, sowie die L1-Caches $L1_1$ und $L1_2$ zu je 100% genutzt. Der $L2_1$ Cache wird von der Belegung „Rot“ zu 50% genutzt. Die Belegung auf Abbildung 2.25(b) ist dagegen nicht valide, da der Knoten $L2_1$ einen $usageSetNode$ Wert größer als 1 hat, da beide Belegungen („Rot“ und „Grün“) ihn zu je 60% beanspruchen.

2.3.1.2 Bestimmung des HW-Graphen

Zur Bestimmung des HW-Graphen existieren mehrere Möglichkeiten, nachfolgend sollen drei von ihnen vorgestellt werden. Eine Möglichkeit ist es, den System- bzw. Datenbankadministrator damit zu beauftragen, diesen Graphen zu erstellen. Das Problem hierbei ist, dass zur Erstellung des Graphen ein detailliertes Wissen der verwendeten Hardware notwendig ist und der Trend zur automatischen Administration [165, 35, 110, 141, 6, 5] von RDBMSen gegen diese Wahl spricht.

Eine andere Möglichkeit ist es, die einzelnen Knoten und Kanten direkt aus der Hardware bzw. über das OS zu bestimmen. Diese Möglichkeit besteht zumindest bei Intel- bzw. AMD-Prozessoren, beide verwenden dazu den *cpuid* Befehl, welcher je nach Aufruf unterschiedliche Parameter und Eigenschaften des Prozessors, inklusive Cache-Eigenschaften, zurück gibt. Unter dem Linux-Betriebssystem können mit Hilfe des Befehl *numactl* detaillierte Informationen über die NUMA-Konfiguration extrahiert werden, dazu kommt die Datei */proc/cpuinfo*, die Auskunft über vorhandene Prozessoren liefert.

Die letzte Möglichkeit besteht darin, das System selbst zu testen und darüber den Graphen zu erstellen. Für diesen Zweck hat Manegold ein Werkzeug namens „Calibrator“ erstellt [113], das die einzelnen Cachestufen und ihre Parameter identifizieren kann. Allerdings wurde es für einen Einzelkern-Prozessor entworfen, es ist somit nicht möglich Caches zu identifizieren, die von mehreren Hardware-Threads geteilt werden, ebenso ist eine Identifikation von ccNUMA-Systemen oder SMT nicht möglich. Im Folgenden sollen ein Vorgehensmodell beschrieben werden, mit dem der komplette HW-Graph erstellt werden kann. Dabei wird der bestehende Calibrator in dieser Arbeit erweitert, um gemeinsame Caches, ccNUMA-Konstellationen und SMT-Kerne zu erkennen.

2.3.1.3 Vorgangsmodell zur Bestimmung des HW-Graphen

Zur Bestimmung des HW-Graphen mit Hilfe des erweiterten Calibrators sind die folgenden Schritte notwendig.

1. Bestimmung der Anzahl von vorhandenen Hardware-Threads n.
2. Identifizierung der Hardware-Threads, die innerhalb eines Kerns durch SMT erzeugt werden.
3. Bestimmung der Cachehierarchien und ihrer Eigenschaften für jeden Hardware-Thread.

4. Erkennung von gemeinsamen Caches.

5. Zerlegung des zentralen Hauptspeicherknotens in die einzelnen ccNUMA-Knoten.

1. Anzahl der Hardware-Threads: Die Zahl n der vorhandenen Hardware-Threads lässt sich meist durch das Betriebssystem bestimmen. Ist dies nicht möglich, kann die Anzahl wie folgt bestimmt werden: (i) Im ersten Schritt wird ein Thread erzeugt, der für eine bestimmte Zeit rechnet. Er sollte dabei möglichst wenig Speicheroperationen durchführen, so dass sich später nicht die einzelnen Threads wegen des Speichers oder der Bandbreite behindern. Eine gute Möglichkeit ist z.B. die Bestimmung von π oder der Fibonacci Zahl für eine große Anzahl von Iterationen. (ii) Im Anschluss wird die Anzahl der Threads stetig um eins erhöht und die gesamte Ausführungszeit aller Threads gemessen. Sobald die Anzahl der Threads die Anzahl der vorhandenen Hardware-Threads übersteigt, wird sich ein starker Sprung in den Ausführungszeiten ergeben.

Abbildung 2.26 zeigt die graphische Auswertung der Durchführung des ersten Schritts auf dem Testsystem (s. Abschn. 2.1.5.4), darauf ist deutlich ein sprunghafter Anstieg der Laufzeit zu erkennen, sobald mehr Threads (≥ 17) als Hardware-Threads vorhanden sind. Damit kann die Knotenmenge $\mathcal{P}$ aller Hardware-Threads des Systems bestimmt werden.

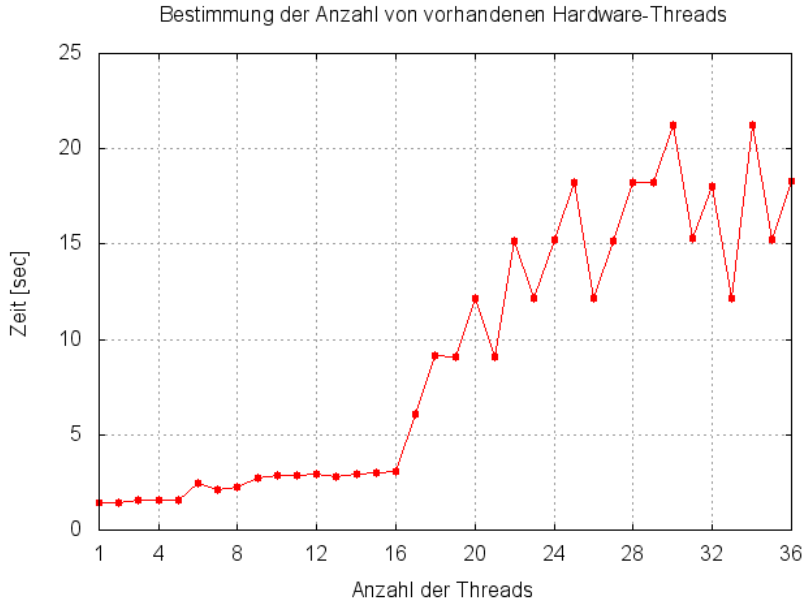


Abbildung 2.26: Auswertung Vorgehensmodell erster Schritt

2. SMT-Kerne: Zur Bestimmung von Hardware-Threads, die sich einen Kern teilen, wird wie folgt vorgegangen: Für alle Paare P_i, P_j aus $\mathcal{P} = \{P_1, ..., P_n\}$ mit $i = 1...n-1$ und $j = i+1...n$ werden zwei Threads kreiert, an die entsprechenden Hardware-Threads (P_i, P_j) gebunden und ihre gesamte Ausführungszeit, bzgl. einer ähnlichen Aufgabe zu

Schritt 1, gemessen. Eine überdurchschnittliche Abweichung von den Einzelausführungszeiten lässt auf SMT-Kerne schließen.

3. Cachehierarchie der einzelnen Hardware-Threads; Zur Bestimmung der Cachehierarchie unterhalb jedes Hardware-Threads muss der Calibrator, in seiner ursprünglichen Form, für jeden einzelnen Hardware-Thread einmal laufen. Daraus kann dann bereits ein HW-Graph erstellt und die entsprechenden Eigenschaften zugewiesen werden. Dieser Graph enthält noch keine Informationen über gemeinsame Caches oder ccNUMA-Informationen, daher könnten vorhandene unterschiedliche Cache-Knoten des Graphen im realen System den gleichen Cache darstellen.

4. Gemeinsame Caches; Zur Bestimmung der gemeinsamen Caches wird ähnlich zur SMT-Kern-Erkennung vorgegangen. Für jedes Paar P_i, P_j mit $i = 1...n-1$ und $j = i+1...n$ wird der Calibrator gleichzeitig auf P_i und P_j ausgeführt. Unterschiedliche Werte bzgl. der einzelnen Cachegrößen aus Schritt drei lassen auf die gemeinsame Nutzung des jeweiligen Caches schließen. Im HW-Graph müssen daher die entsprechenden Cache-Knoten entfernt und durch einen neuen gemeinsamen Knoten ersetzt werden.

5. ccNUMA: Um ccNUMA zu erkennen, d.h., der Hauptspeicher Knoten im HW-Graphen stellt eigentlich eine Menge von Knoten dar, wird wie folgt vorgegangen: (i) Wie zuvor wird paarweise für die Hardware-Threads P_i und P_j mit $i = 1...n-1$ und $j = i+1...n$ je ein Thread T_i und T_j erzeugt (Kombinationen, die einen gemeinsamen Cache mit HW-Graphen haben, können übersprungen werden). (ii) Danach alloziert der Thread T_i auf P_i einen Bereich M an Speicher. (iii) Dann führt T_i Lese- und Schreiboperationen auf M aus und misst die benötigte Zeit. (iv) Anschließend führt T_j Lese- und Schreiboperationen auf M aus und vergleicht seine Zeit mit der von T_i. Auch hier ist ein überdurchschnittlicher Unterschied, speziell eine Verschlechterung, als Indikation für ccNUMA anzusehen, und der HW-Graph muss daraufhin entsprechend angepasst werden.

Im Zuge der Evaluierung wurden auch die restlichen Schritte ausgeführt und der HW-Graph auf Abbildung 2.24(c) ermittelt.

2.3.2 Operatoren des Hardwaremodells

Die Operatoren des Hardwaremodells sollen einerseits eine bessere Trennung zwischen der Datenbankseite und der Hardwareseite bringen; andererseits sollen sie zu einem besseren Leistungsverhalten führen, da die gegebene Hardware und ihre speziellen Eigenschaften besser genutzt werden.

Es werden drei Arten von Operatoren unterschieden: (i) Bibliotheksoperatoren, (ii) statische Laufzeitoperatoren und (iii) dynamische Laufzeitoperatoren. Im Folgenden soll jede Art an einem konkreten Beispiel vorgestellt bzw. motiviert werden. Mit dieser Arbeit soll dabei keine abgeschlossene Menge von Operatoren entworfen werden, vielmehr werden einzelne Operatoren, die in der Arbeit benötigt werden, vorgestellt. Da die Realisierung von der Rechnerarchitektur abhängt, werden die Operatoren in restlichen Teil dieser Arbeit auch als abstrakte Operatoren bezeichnet, der Begriff „abstrakt“ soll dabei

die Architekturunabhänigkeit widerspiegeln.

2.3.2.1 Bibliotheksoperatoren

Bibliotheksoperatoren sind der Teil der abstrakten Operatoren, die bzgl. einer Mikroarchitektur nur eine bestimmte optimale Ausführung haben. Sie werden durch Bibliotheken, die an das jeweilige Rechnerarchitektur angepasst sind, ausgeliefert. Dazu solle das Beispiel 2.11 nochmals aufgegriffen werden.

Mit dem abstrakten Bibliotheksoperator **BitCount** kann die Aufgabe aus dem Beispiel deutlich erleichtert werden, indem der abstrakte Operator *BitCount* anstelle eines Algorithmus oder des Hardwarebefehls verwendet wird. Die jeweilige Implementierung übernimmt der Hardwareexperte und stellt sie in einer Bibliothek zur Verfügung, in der der abstrakte Operator implementiert ist. Auf einem gegebenen Rechnersystem wird somit immer ein und dieselbe Funktion ausgeführt, wenn der *BitCount*-Operator aufgelöst wird, nämlich die optimale Lösung für die jeweilige Architektur.

In diesem Beispiel wird davon ausgegangen, dass alle Kerne im System die gleiche ISA implementieren; sollten einzelne Kerne unterschiedliche Befehle implementieren, müssen diese Befehle durch dynamische Laufzeitoperatoren dargestellt werden.

2.3.2.2 Statische Laufzeitoperatoren

Neben den Problemen, für die eine generelle optimale Lösung existiert, sind viele Lösungen auch kontextabhängig. Für diese Lösungen existieren **statische Laufzeitoperatoren**, die ebenfalls an einem Beispiel erläutert werden.

Beispiel 2.14: Statistischer Laufzeitoperator SumList

```
uint32_t sumList(Liste* list, uint32_t count){
    uint32_t sum = 0;
    for(uint32_t i=0; i<count; ++count){
        sum += list.A[count];
    }
    return sum;
}
```

Es sollen alle Zahlen des Elementes A einer Liste aufsummiert werden. Diese Summation wird vom Programm in Beispiel 2.14 geleistet; auf einer entsprechenden Hardware wäre aber eine SIMD-Berechnung (Def. 2.8) möglicherweise schneller. Diese Entscheidung ist abhängig von der Größe des Listenelementes und wie, bzw. wie gut die jeweilige Mikroarchitektur SIMD unterstützt.

Die Klasse der statischen Laufzeitoperatoren ist somit der der Bibliotheksoperatoren sehr ähnlich, allerdings mit dem entscheidenden Unterschied, dass erst zur Laufzeit entschieden wird, welches das beste Vorgehen ist. Zur Laufzeit heißt dabei, sobald die korrekte Instantiierung der Liste bekannt ist. Ist aber einmal die Entscheidung getroffen, wird sie nicht mehr verändert, daher sind es *statische* Laufzeitoperatoren.

2.3.2.3 Dynamische Laufzeitoperatoren

Neben den statischen Laufzeitoperatoren existieren noch **dynamische Laufzeitoperatoren**. Die Funktion oder der Algorithmus, der durch einen **dynamischen Laufzeitoperator** ausgeführt wird, hängt nicht nur von der Laufzeit ab, sondern auch von der aktuellen Belegung für den Operator. Es soll das Verhalten eines dynamischen Laufzeitoperators am Beispiel 2.10 und dem abstrakten Operator *MemTransfer(*Op_1*,*Op_2*)* vorgestellt werden. Mit dem abstrakten Operator *MemTransfer* wird ein Speicheraustausch zwischen zwei Operatoren Op_1 und Op_2 beschrieben. Die Instantiierung ist dabei belegungsabhängig, wie im Beispiel 2.10 beschrieben. Werden die beiden Operatoren auf unterschiedlichen NUMA-Knoten ausgeführt, ist es vielleicht von Vorteil, den kompletten Bereich auf einmal zwischen den Knoten hin und her zu senden. Werden die beiden Operatoren dagegen auf dem gleichen Prozessor ausgeführt, genügt es die Daten zwischen den verschiedenen Caches auszutauschen, wenn eine solche Operation von der Prozessorarchitektur angeboten wird.

2.3.3 Modelle zur Parallelisierung

Im letzten Abschnitt sollen zwei mit dieser Arbeit entwickelte Modelle zur Parallelisierung vorgestellt werden [21, 48, 81]. Beim ersten Modell handelt es sich um ein Modell für parallele Schleifen und Schleifen mit Flussabhängigkeit (s. Abschn. 2.2.2.2), bei dem der Leistungsgewinn über den Grad der Parallelität beschrieben wird. Das zweite Modell dient zur Beurteilung der verschiedenen Sperrmechanismen und soll zur Bewertung und zur Auswahl eines geeigneten Sperrmechanismus dienen.

2.3.3.1 Modelle für parallele Schleifen

Für Schleifen ohne Abhängigkeiten der Größe N mit einer Ausführungszeit t_{it} pro Iterationsschritt und P Hardware-Threads lässt sich der *SpeedUp* (Def. 1.8, Seite 10) bei Parallelisierung wie folgt berechnen:

Definition 2.39 (SpeedUp Parallele Schleifen ohne Abhängigkeiten)

$$SpeedUp(P) = \frac{t_{seriell}}{t_{parFor}} = \frac{Nt_{it}}{\frac{N}{P}t_{it}} = P$$

□

Für eine Schleife mit Flussabhängigkeit (t_{ForDep} -Gesamtausführungszeit) und der zusammen mit dem Beispiel 2.7 beschriebenen Parallelisierung, berechnet sich der *SpeedUp* mit:

Definition 2.40 (SpeedUp Parallele Schleifen Flussabhängigkeit)

$$SpeedUp(N,P) = \frac{t_{seriell}}{t_{ForDep}} = \frac{Nt_{it}}{2\frac{N}{P}t_{it} + (P-1)t_{it}} = \frac{N}{2\frac{N}{P} + P - 1}$$

□

Beide Formeln sind vereinfacht, da angenommen wird, dass die Taskgenerierung keinerlei Zeit in Anspruch nimmt oder gegenüber t_{it} vernachlässigbar ist. Für den Fall,

dass diese Annahme nicht korrekt ist, wurde ein Modell für beide Schleifenarten erstellt werden, welches die Zeit t_{task} für die Taskgenerierung und Taskabarbeitung einbezieht.

Die Zeit für die serielle Abarbeitung bleibt wie zuvor, die Zeiten für die Ausführung der parallelen Schleife ohne und mit Flussabhängigkeit ändern sich wie folgt:

$$\begin{aligned} t_{parFor} =& \frac{N}{P} t_{it} + P t_{task} \\ t_{ForDep} =& \frac{2N}{P} t_{it} + (P-1) t_{it} + 2P t_{task} \end{aligned}$$

Der optimale Parallelisierungsgrad bestimmt sich mit:

$$\left(\frac{N}{P} t_{it} + P t_{task} \right) \frac{d}{dP}$$

bzw.

$$\left(\frac{2N}{P} t_{it} + (P-1) t_{it} + 2P t_{task} \right) \frac{d}{dP}$$

Definition 2.41 (optimaler Parallelisierungsgrad) Der optimalen Parallelisierungsgrad P_{opt} einer parallelen Schleife ist definiert als:

$$P_{opt} = \sqrt{N \frac{t_{it}}{t_{task}}}$$

Für eine Schleife mit Flussabhängigkeit gilt:

$$P_{opt} = \sqrt{\frac{N}{0.5 + \frac{t_{task}}{t_{it}}}}$$

□

Über den optimalen Grad der Parallelität lässt sich dann die optimale Partitionsgröße (s. Seite 60) mit $\frac{N}{P_{opt}}$ bestimmen.

2.3.3.2 Modell für Sperrmechanismen

Im Folgenden soll ein Modell zur Beurteilung von Sperrmechanismen vorgestellt werden [36]. Darin wird davon ausgegangen, dass ein Thread T_i eine bestimmte Sperre N mal durchläuft und dabei die Zeit t pro Durchlauf benötigt. Die Zeit t setzt sich aus folgenden Teilzeiten zusammen:

$t = t_L + t_W + t_S = t_{LG} + t_{LW} + t_W + t_S$
$t_L = t_{LG} + t_{LW}$: T_i hält die Sperre
t_{LG} : der Teil von t_L zum Setzen und Freigeben der Sperre
t_{LW} : der Teil von t_L in dem T_i auf gemeinsamen Variablen arbeitet
t_W : T_i hält die Sperre nicht, verrichtet aber sinnvolle Arbeit
t_S : T_i wartet auf die Sperre

Definition 2.42 (*LockLoss*) Der Leistungsverlust *LockLoss* durch eine Sperre berechnet sich dann aus dem Quotienten der Zeit mit Sperre und der ohne Sperre ($t_S = 0, t_{LG} = 0$):

$$LockLoss = \frac{N(t_{LG} + t_{LW} + t_W + t_S)}{N(t_{LW} + t_W)} = 1 + \frac{t_{LG} + t_S}{t_{LW} + t_W}$$

Während sich t_{LG}, t_{LW} und t_W einfach aus dem System bestimmen lassen, kann t_S durch folgende Überlegung ersetzt werden: t_S ist die Zeit, die der Thread T_i auf die Sperre warten muss. Dabei wird bei P Threads die Sperre pro Durchlauf von jedem Thread für die Zeit t_L gehalten und somit kann der Thread T_i die Sperre für einen Zeitraum von $(P-1)t_L$ nicht erlangen. Stattdessen kann er aber in dieser Zeit die Arbeit ausführen, die keine Sperre benötigt. Dazu benötigt er t_W und daraus ergibt sich:

$$t_S = \begin{cases} 0 & \text{if } t_W \geq (P-1)t_L, \\ (P-1)t_L - t_W & \text{sonst.} \end{cases}$$

$$LockLoss = 1 + \begin{cases} \frac{t_{LG}}{t_{LW}+t_W} & \text{if } t_W \geq (P-1)t_L, \\ \frac{t_{LG}+(P-1)t_L-t_W}{t_{LW}+t_W} & \text{sonst.} \end{cases}$$

□

Aus diesen Überlegungen lässt sich auch der *SpeedUp* der Parallelisierung errechnen, wobei jeder Thread die Sperre $N = \frac{k}{P}$ mal durchläuft.

$$\begin{aligned} SpeedUp &= \frac{k(t_{LW} + t_W)}{\frac{k}{P}(t_L + t_W + t_S)} \\ SpeedUp_{t_S=0} &= P\frac{t_{LW} + t_W}{t_L + t_W} \\ SpeedUp_{t_S>0} &= \frac{t_{LW} + t_W}{t_L} \end{aligned}$$

Angenommen, dass nur t_S von P abhängt und zwar linear wie oben beschrieben, berechnet sich der maximalen *SpeedUp* mit $\frac{t_{LW}+t_W}{t_L}$ bei einem maximal sinnvollen Parallelisierungsgrad von $P_{opt} = 1 + \frac{t_W}{t_L}$. Diese Annahme ist in sofern nicht korrekt, da

zumindest die Zeit für die Sperrmodifikation t_{LG} und meist auch t_{LW} von P abhängt (speziell unter Cache-Kohärenz Abschn. 2.1.1.2). Sei also $t_{LG} = t_{LG1} + Pt_{LG2}$, so gilt auch in diesem Fall, dass der Übergang von $t_S = 0$ zu $t_S > 0$ den maximalen Parallelitätsgrad definiert.

Evaluierung des Modells für Sperrmechanismen

Zuletzt wird das Modell für Sperrmechanismen auf seine Praxistauglichkeit geprüft. Auf Abbildung 2.27 ist jeweils die Modellvorhersage und das reale Verhalten für eine unterschiedliche Frequentierung t_w (1000, links / 7000, rechts) für ein Betriebssystemmutex (Mutex) und einen kritischen Bereich (engl. Critical Section) (CritSec) dargestellt. Dabei wurde für die Berechnung der Zeit für t_{LG} die Formel $t_{LG} = t_{LG1} + Pt_{LG2}$ genutzt. Es ist zu sehen, dass das Modell mit Hilfe der Parameter aus Tabelle 2.4 die Realität gut beschreibt. Die Messungen wurden auf dem Testsystem (s. Abschn. 2.1.5.4) mit Hilfe der in der Windows-API zur Verfügung gestellten Sperrmechanismen *CreateMutex* - Mutex und *InitializeCriticalSection* - kritischer Bereich durchgeführt.

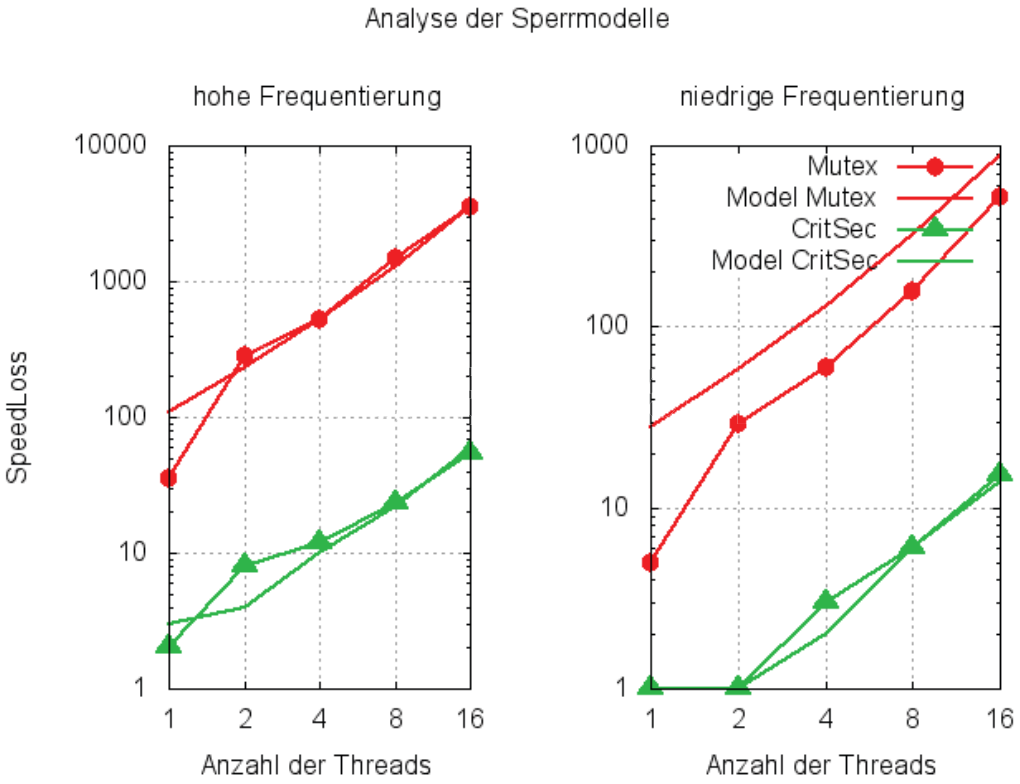


Abbildung 2.27: Evaluierung des Sperrmodells

Parameter	Mutex	CritSec
t_{LG1}	200000	3000
t_{LG2}	15000	200
t_{LW}	1000	1000

Tabelle 2.4: Verwendete Werte für das Modell

Zusammenfassung

In diesem Kapitel wurde im ersten Abschnitt der Aufbau von Speicherhierarchien, Caches und Prozessoren dargestellt. Ergänzt wurde diese Beschreibung durch Möglichkeiten der Parallelität auf Instruktionsebene, bei denen besonders SIMD (Def. 2.8) hervorzuheben ist. Weiter wurden die Prinzipien von Mehrkern-Rechnerarchitekturen (Def. 2.11) zusammen mit Unterschieden zu Mehrprozessor-Systemen vorgestellt. Am Ende des Abschnitts wurden verschiedene Mehrkern-Prozessoralternativen der nächsten Generation und ihre Eigenschaften diskutiert.

Im zweiten Abschnitt wurde zuerst motiviert, warum es notwendig ist, parallele Programme zu schreiben. Es wurden parallele Leistungsmaße eingeführt und im Anschluss Techniken der parallelen Programmierung vorgestellt. Darunter das Konzept von Tasks (Def. 2.20), welches ein gutes Leistungsverhalten auf Mehrkern-Prozessoren verspricht. Gefolgt von Zusammenhängen von Programmen, Prozessen, Threads und verschiedenen Möglichkeiten der Threadausführung. Den Abschluss bildete die Vorstellung und der Vergleich von verschiedenen parallelen Programmierumgebungen, darunter Intels Thread-Building-Blocks und Microsofts ConcRT, die beide auf dem Konzept von Tasks beruhen und Frameworks für die parallele Programmierung mit der Programmiersprache C++ darstellen.

Im dritten und letzten Teil wurde der erste Teil des Frameworks zur Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen vorgestellt. Dabei wurde ein Hardwaremodell definiert, das über den Hardware-Graphen dargestellt und genutzt werden kann. Aufbauend auf diesem Hardwaremodell wurden im Anschluss verschiedene Klassen von abstrakten Operatoren vorgestellt, die es ermöglichen, die vorhandene Hardware besser zu nutzen und effizienter zu entwickeln. Zum Abschluss wurden zwei Modelle zur Parallelisierung vorgestellt, die der Bestimmung des optimalen Parallelisierungsgrads bzw. der Granularität von parallelen Schleifen ohne und mit Flussabhängigkeit dienen, sowie für die Beurteilung und die Auswahl von verschiedenen Sperrmechanismen genutzt werden können.

3 Das physische Datenmodell

„Der Ziellose erleidet sein Schicksal, der Zielbewußte gestaltet es.“

Marcus Tullius Cicero (106–43 v. Chr.)

Im dritten Kapitel dieser Arbeit wird das physische Datenmodell die zentrale Rolle spielen. Zuerst werden die konzeptuelle, danach die interne Schicht der 3-Schichten-Architektur (Def. 1.3) näher betrachtet. Da sich im Fokus dieser Arbeit RDBMSe befinden, wird das relationale Modell als Datenmodell der konzeptionellen Schicht definiert. Darauf aufbauend wird zunächst das traditionelle, zeilenorientierte Datenmodell vorgestellt, dem die Definition des spaltenorientierten Datenmodells folgt. Beide stellen Konzepte zur Repräsentation der konzeptuellen Schicht auf die internen Schicht dar und sollen bzgl. der Verwendung und Eignung als physisches Datenmodell im Rahmen dieser Arbeit verglichen werden. Im vorletzten Teil des Kapitels wird die Schicht des physischen Datenmodells des in dieser Arbeit entwickelten Frameworks (s. Abschn. 1.3) präsentiert. Im Anschluss wird das Konzept der globalen Identifikatoren definiert und eine mögliche Einbettung in das physische Datenmodell vorgestellt. Des Weiteren wird gezeigt welche Nutzungsmöglichkeiten und Vorteile die Verwendung von globalen Identifikatoren bieten. Den Abschluss des Kapitels stellt der Abschnitt über ein in dieser Arbeit entwickelter Index dar, der die Möglichkeiten von Leistungsverbessungen durch SIMD-Operationen (Def. 2.8) nutzen kann.

3.1 Das Relationenmodell

Das Relationenmodell wurde 1970 von Ted Codd [38] entwickelt. Es basiert auf dem Konzept von mathematischen **Relationen** mit folgender Definition:

Definition 3.1 (Relation) Die **Domänen** $D_1, D_2, ..., D_n$ seien gegeben. Ferner dürfen sie nur atomare und nicht strukturierte Werte enthalten. Eine **Relation** R ist dann als Teilmenge des kartesischen Produkts der n Domänen definiert:

$$R \subseteq D_1 \times D_2 \times ... \times D_n$$ □

Die Relation R kann bzgl. ihrer **Extension**, also seiner aktuellen Ausprägung, bzw. ihrer **Intension**, ihrem Schema, unterschieden werden. Das Schema von R (kurz $\mathbf{sch}(R)$

oder $\mathcal{R}$) ist die Liste all ihrer Attribute. Die Domäne des Attributes A_i wird mit $\mathbf{dom}(A_i)$ bezeichnet. Die dem **Relationenschema** $\mathcal{R} = (A_1, A_2, ..., A_n)$ zugehörige Relation R ist dann definiert als:

$$R \subseteq \mathbf{dom}(A_1) \times \mathbf{dom}(A_2) \times ... \times \mathbf{dom}(A_n)$$

Elemente der Extensionsmenge von R werden als **Tupel** bezeichnet. Jedes Tupel kann dabei als eine Abbildung eines Attributs auf seinen Domänenwert aufgefasst werden:

$$\begin{aligned} \mu : \{A_1 ... A_n\} \rightarrow & D_1 \cup D_2 \cup ... \cup D_n, \text{mit} \\ \mu(A_i) \rightarrow & d \in D_i \end{aligned}$$

Die Verallgemeinerung der Abbildung auf eine Attributmenge S ist dann definiert als:

$$\begin{aligned} S = & \{A_{j_1}, ..., A_{j_k}\} \subseteq \{A_1, ..., A_n\} \\ \mu[S] = \mu[\{A_{j_1}, ..., A_{j_k}\}] = & d_{j_1} \in \mathbf{dom}(A_{j_1}) \times ... \times d_{j_k} \in \mathbf{dom}(A_{j_k}) \end{aligned}$$

Um auf dem definierten Datenmodell operieren zu können, wird eine Sprache benötigt. Für das Relationenmodell existieren unter anderen folgende zwei formalen Sprachen: (i) Die **relationale Algebra** und (ii) das **Relationenkalkül**, das eine rein deklarative Sprache ist. Die relationale Algebra ist dagegen stärker prozedural orientiert.

Im Rahmen dieser Arbeit wird auf Operatoren einer erweiterten relationalen Algebra zurückgegriffen, um SQL-Anfragen in eine äquivalente baumartige prozedurale Form zu transformieren [150]. Es seien $R(A_1, A_2, ..., A_n)$ und $S(B_1, B_2, ..., B_m)$ zwei Relationen, dann sind folgende Operatoren definiert:

(i) **Selektion σ:** Die Selektion $\sigma_F(R)$ bzgl. des prädikatenlogischen Ausdrucks F auf der Relation R wählt alle Tupel aus R aus, die den Ausdruck F erfüllen.

$$R' = \sigma_F(R) = \{\mu \mid \mu \in R \wedge F[\mu] = wahr\}$$

(ii) **Projektion π:** Die Projektion $\pi_P(R)$ bzgl. der Attributmenge P auf der Relation R bildet alle Tupel aus R in das neue Schema mit den Attributen aus P ab.

$$\begin{aligned} P = & \{A_{j_1}, ..., A_{j_k}\} \subseteq \{A_1, ..., A_n\} \\ R' = & \pi_P(R) = \{\mu[P] \mid \mu \in R\} \end{aligned}$$

(iii) **Join $\bowtie$:** Der Join $\bowtie_C (R, S)$ bzgl. des Ausdrucks C auf den Relationen R und S, verknüpft jedes Tupel aus R mit jedem Tupel aus S, bei denen die Joinbedingung C erfüllt ist.

$$\begin{aligned} T = & \bowtie_C (R, S) \\ = & \{\mu = \mu_R \times \mu_S \mid \mu_R \in R \wedge \mu_S \in S \wedge C[\mu] = wahr\} \end{aligned}$$

Für den Spezialfall des Naturaljoins wird die Joinbedingung C weggelassen und ist implizit darüber definiert, dass eine paarweise Gleichheit aller gemeinsamen Attribute in R und S gefordert wird.

$$\begin{aligned} P =& \{A_{j_1}, ..., A_{j_k}\} = sch(R) \cap sch(S) \\ C =& (R.A_{j_1} = S.A_{j_1} \wedge ... \wedge R.A_{j_k} = S.A_{j_k}) \\ T =& \bowtie (R, S) \\ =& \{\mu = \mu_R \times \mu_S \mid \mu_R \in R \wedge \mu_S \in S \wedge C[\mu] = wahr\} \end{aligned}$$

(iv) **Umbenennung ρ:** Die Umbenennung $\rho_{S(B_1,...,B_n)}(R)$ erstellt eine zu R identische Relation mit dem Namen S. Dabei werden die Attribute A_i mit $i = 1, ..., n$ in B_i umbenannt.

(v) **Gruppierung γ:** Die Gruppierung $\gamma_F^A(R)$ stellt Gruppen bzgl. der Attributmenge A auf der Relation R dar. Des Weiteren werden auf jeder Gruppe die Aggregatfunktionen aus F ausgewertet. Übliche Aggregatfunktionen sind: *count, sum, max, avg.*

$$\begin{aligned} A =& \{A_{j_1}, ..., A_{j_k}\} \subseteq \{A_1, ..., A_n\} \\ B_i =& \{B_{i_1}, ..., B_{i_l}\} \subseteq \{A_1, ..., A_n\} \\ F =& \{f_1(B_1), ..., f_m(B_m)\} \\ R' =& \gamma_F^A(R) \\ =& \{d_1 \times ... \times d_k \times f_1(\bigcup \mu_B[B_1]) \times ... \times f_m(\bigcup \mu_B[B_m]) \mid \\ & \forall d_1, ..., d_k \in \mathbf{dom}(A_{j_1}) \times ... \times \mathbf{dom}(A_{j_k}) \wedge \\ & \exists \mu_B \in R \wedge \mu_B[A] = d_1, ..., d_k\} \end{aligned}$$

(vi) **Sortierung τ:** Die Sortierung $\tau_L(R)$ bzgl. der Attributmenge L auf der Relation R, überführt die Menge der Tupel von R in eine bzgl. L geordnete Liste von Tupel. Da das Ergebnis keine Relation mehr darstellt, darf der Sortierungsoperator nur am Ende stehen und von keinem weiteren Operator gefolgt werden.

Beispiel 3.1: TPC-H-Anfrage Q_5

```
select n_name,
   sum(l_extendedprice * (1 - l_discount)) as revenue
from customer, orders, lineitem, supplier, nation, region
where c_custkey = o_custkey and l_orderkey = o_orderkey
   and l_suppkey = s_suppkey and c_nationkey = s_nationkey
   and s_nationkey = n_nationkey and n_regionkey = r_regionkey
   and r_name = '[REGION]' and o_orderdate >= date '[DATE]'
   and o_orderdate < date '[DATE]' + interval '1' year
group by n_name order by revenue desc;
```

Abbildung 3.1 zeigt die obige SQL-Anfrage in ihrer Operatorbaumdarstellung unter Verwendung der Operatoren der erweiterten relationalen Algebra.

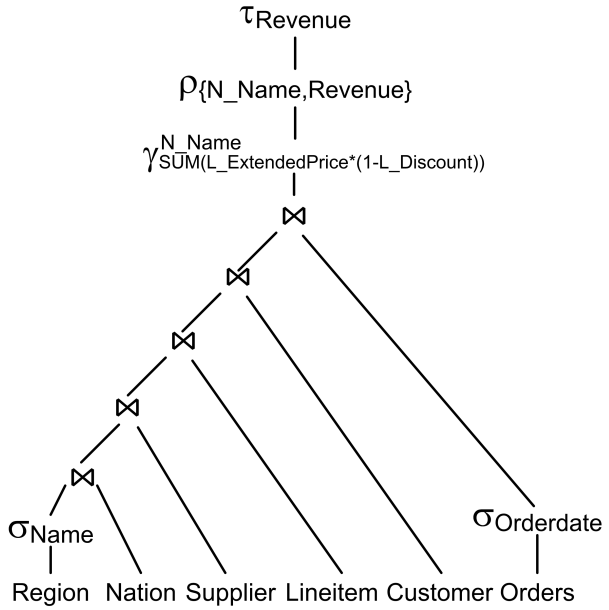


Abbildung 3.1: TPC-H-Anfrage Q_5 in der erweiterten relationalen Algebra

3.2 Das physische Datenmodell

Im Abschnitt 1.2 wurde bereits die 3-Schichten-Architektur als Mittel zur Datenabstraktion in RDBMSen vorgestellt (Def. 1.3). Das Relationenmodell stellt die konzeptuelle Schicht dar, und die interne Schicht wird durch das physische Datenmodell umgesetzt. Dazu kommen die Möglichkeiten der Abbildung des Relationenmodells auf das physische Modell und vice versa.

Für das physische Datenmodell existieren verschiedene Konzepte, die sich durch die Anordnung der Informationen der einzelnen Tupel unterscheiden [42, 105, 7, 25, 66, 65]. Im Folgenden wird zunächst die **horizontale Partitionierung**, dann die **vertikale Partitionierung** und abschließend eine hybride Form der beiden vorhergehenden vorgestellt. Dazu kommt eine nähere Betrachtung der Umsetzung der vertikalen Partitionierung in MonetDB, um später im Kapitel 4 darauf aufbauen zu können. Das Ende des Abschnitts stellt eine kritische Bewertung und der Vergleich der verschiedenen Modelle dar und dient im Abschnitt 3.3 als Basis für die Auswahl eines physischen Datenmodelle für das Framework.

3.2.1 Horizontale Partitionierung

Beim physischen Datenmodell der **horizontalen Partitionierung**, auch N-näres Speichermodell (NSM) genannt [7], werden die Daten wie in der konzeptuellen Schicht abgespeichert. Alle Datenwerte eines Tupels, also alle n Werte der n Attribute, werden zusammen bzw. hintereinander im Speicher angeordnet. Dazu kommt meist noch für jedes einzelne Tupel ein sogenannter **Tupelheader**, welcher unter anderem Informationen zur Länge der einzelnen Attributwerte und über eventuelle `NULL` Werte bereitstellt.

NSM – N-nary Storage Model

Beispiel 3.2: Beispiel zur horizontalen Partitionierung

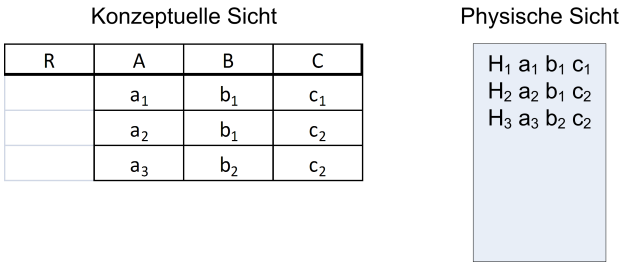


Abbildung 3.2: Horizontalen Partitionierung

Auf Abbildung 3.2 ist ein Beispiel für die Abbildung der konzeptuellen Schicht (links) in das physische Datenmodell der horizontalen Partitionierung (rechts) dargestellt. Der grau hinterlegte Bereich soll einen Abschnitt des Hauptspeichers oder eine Seite des Sekundärspeichers darstellen (s. Abschn. 2.1.1). Die Symbole H_i stehen für den Header des i-ten Tupels, sowie die Attributwerte a_x, b_y, c_z mit den entsprechenden Werten der linke Seite korrespondieren.

Das Datenmodell der horizontalen Partitionierung wird meist noch um (interne) Datenstrukturen ergänzt, die die Anfragebearbeitung beschleunigen sollen. **Indizes** dienen dazu Anfragen oder Teile von Anfrage zu beantworten, bei denen durch ein Prädikat nur ein geringer Teil der Gesamtrelation für die Antwort infrage kommt. Das entsprechende Suchkriterium, auf dem ein Index definiert ist, wird als **Schlüssel** bezeichnet [103].

Definition 3.2 (Index) Ein **Index** ist eine Datenstruktur, die bzgl. eines Suchkriteriums Q mit den Schlüsselwerten $q_1, ..., q_n$ definiert ist und jedem Schlüssel q_i einer Menge von Tupeln zuordnet, die alle das Suchkriterium mit q_i erfüllen. □

Für das Konzept des Indexes existieren eine Vielzahl von möglichen Realisierungen, darunter indexsequentielle Dateien und B-Bäume. Für eine komplette Darstellung sei auf Abschnitt 7.5 in [103] verwiesen. Im Folgenden sollen noch zwei spezielle Formen eines Index definiert werden, der Domänenindex und der Joinindex. Während Indizes im Allgemeinen auf einer einzelnen Relation definiert sind, stellt der Domänenindex eine Indexstruktur über mehrere Relationen der Datenbank dar.

Definition 3.3 (Domänenindex) Bei einem **Domänenindex** ist das Suchkriterium auf einer Domäne D – nicht auf einem Attribute – definiert, d.h., die Schlüsselwerte werden durch die Gesamtheit aller in der Datenbank existierenden Attributwerte bzgl. dieser Domäne definiert. Der Domänenindex ordnet jedem Schlüsselwert eine Menge von Tupel zu, die das jeweilige Suchkriterium erfüllen. Die Tupel können dabei aus unterschiedlichen Relationen stammen [130]. □

Eine aufwendige Operation während der Anfrageausführung stellen Joins dar. Sattler merkt in [147] an „Eine besondere Bedeutung kommt den effizienten Join-Operationen zu ...“; zu Zweck der effizienten Joinverabeitung werden Joinindizes verwendet.

Definition 3.4 (Joinindex) Ein Joinindex repräsentiert das Ergebnis eines Joins zwischen mehreren Relationen. Im allgemeinen sind Joinindizes auf zwei Relationen definiert und stellen das Ergebnis eines Equijoins dar. □

3.2.2 Vertikale Partitionierung

DSM – Decomposition Storage Model

Die **vertikale Partitionierung** bzw. das Zerlegungs-Speichermodell (DSM) speichert Tupel der konzeptuellen Schicht, indem jedes Tupel in seine Attributwerte zerlegt wird und diese dann separat gespeichert werden. Auf Grund dieser Eigenschaft wird es auch oft **spaltenorientiertes** Datenmodell genannt, im Gegensatz dazu wird auf NSM mit **zeilenorientiertes** Datenmodell referenziert.

Copeland et al. definieren in [42] das DSM als eine Zerlegung der originalen Relation $R(S, A_1, ..., A_n)$ in n binäre Relationen $R_i(S, A_i)$, wobei S ein Surrogate ist.

Definition 3.5 (Surrogate) Als **Surrogate** werden Attribute bezeichnet, die während der Modellierung hinzugefügt wurden, um als Primärschlüssel zu dienen [168, 41]. Dabei werden die Werte für das Surrogate nicht von der Entität – dem Tupel – bestimmt, sondern werden meist durch das RDBMS generiert (z.B. durch inkrementelle Zähler) und enthalten somit keine für das Tupel wichtigen Daten. □

Surrogate ersetzen dabei andere, meist zusammengesetzte Schlüsselkandidaten und oder Schlüsselkandidaten auf Domänen, die keine effizienten Vergleichsoperationen haben, wie z.B. Zeichenketten und sollen damit speziell bei der Joinberechnung die Anfragebearbeitung effizienter gestalten.

Im DSM werden nur die einzelnen binären Relationen R_i gespeichert. Um einen schnelleren Zugriff zu ermöglichen, wird jede binäre Relation R_i in zwei Formen materialisiert. Zum einen wird R_i auf dem Surrogate S sortiert gespeichert, zum anderen auf den Attributwerten des Attributes A_i. Dieser Teil stellt letztlich einen Index auf dem Attribut A_i bzgl. der Relation R dar. Während der Anfragebearbeitung wird die originale Relation R aus den einzelnen binären Relationen R_i mit Hilfe des Naturaljoins $R = R_1 \bowtie ... \bowtie R_N$ zurückgewonnen.

Beispiel 3.3: Beispiel zur vertikalen Partitionierung

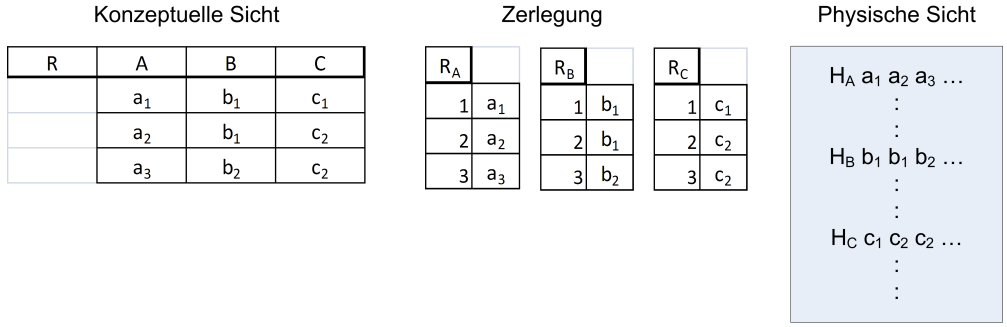


Abbildung 3.3: Vertikalen Partitionierung

Abbildung 3.3 stellt die Abbildung der Relation R in die physische Schicht mit Hilfe der vertikalen Partitionierung dar. Wie in Beispiel 3.2 ist links die konzeptuelle Sicht der Relation R dargestellt und rechts die Repräsentation mit Hilfe des physischen Datenmodells. Die Abbildung in der Mitte zeigt die Zerlegung von R in die einzelnen Teilrelationen R_i. Im Unterschied zum Beispiel 3.2 ist zusehen, dass alle Attributwerte eines Attributes hintereinander angeordnet sind. Die Header H sind nun keine Tupelheader mehr, sondern Attributheader, jede Attributspalte besitzt einen solchen Header.

Das DSM als physisches Datenmodell wurde aufgrund des zusätzlichen Aufwands bei der Rekonstruktion der Ausgangsrelation zuerst keine große Beachtung geschenkt [7], so dass bis Mitte der 90iger Jahre außer Sybase-IQ [160] alle großen RDBMSe NSM als physisches Datenmodell implementiert hatten. Erst mit MonetDB [27] und später mit C-Store [156] wurde das DSM wiederbelebt und ist speziell im Data-Warehouse-Bereich (Def. 1.1) sehr populär geworden.

MonetDB

CWI – Centrum Wiskunde & Informatica

MonetDB ist ein spaltenorientiertes DBMS, das am CWI an der Universität von Amsterdam in der Gruppe von Martin Kersten entwickelt wird. In dieser Arbeit soll MonetDB als Grundgerüst für Teile der prototypischen Implementierung genutzt werden. Aus diesem Grund wird das physische Datenmodell von MonetDB im Folgenden etwas detaillierter betrachtet und später in Abschnitt 4.2.2 das Anfrageausführungsmodell von MonetDB vorgestellt.

Die interne Datenrepräsentation von MonetDB ist – im Gegensatz zu herkömmlichen RDBMSen – nicht sekundärspeicher- und damit seitenorientiert, sondern vielmehr hauptspeicherorientiert. Dazu werden die Inhalte der Dateien vom Sekundärspeicher direkt über das Betriebssystem in den Hauptspeicher abgebildet. Das physische Datenmodell von MonetDB basiert auf dem Konzept der *vertikalen Partitionierung*; in der konzeptuellen Schicht können verschiedene Modelle verwendet werden, darunter z.B. das Relationenmodell, Modelle für hierarchische und semistrukturierte Daten (z.B. XML). Diese

Arbeit wird sich nur mit dem Relationenmodell als Modell der konzeptuellen Schicht beschäftigen. Die Grundidee bzgl. des physischen Datenmodells in MonetDB ist, alles in binären Tabellen darzustellen, also Tabellen mit zwei Spalten. Diese Tabellen werden binäre Assoziationstabellen (BATs) genannt und bestehen aus einer sogenannten **Head**-Spalte (links) und einer **Tail**-Spalte (rechts). Die Daten von Head und Tail liegen in jeweils für sich kontinuierlichen Bereichen des Primärspeichers und können vom Typ her, die Basistypen fester Größe wie *boolean*, *char*, *short*, *int*, *long*, *float* und *double* annehmen. Dazu kommen noch Zeichenketten mit variabler Größe und zwei Typen, um Objekte zu referenzieren (*OId* und *VOId*). *VOIds* stellen virtuelle *OIds* dar und werden nicht physisch gespeichert, sondern vielmehr implizit durch die Zeilennummer oder die Position der jeweiligen Zeile in der BAT und einem generellen Offset kodiert.

BAT – Binary Association Table

Bei der Abbildung des Relationenmodells in die physische Schicht wird in MonetDB auf DSM zurückgegriffen. In MonetDB werden allerdings nicht, wie von Copeland et al. [42] vorgeschlagen, zwei unterschiedlich sortierte Formen jeder Zerlegung materialisiert, sondern nur eine. Anfänglich wird dabei die Form gewählt, die auf den Surrogaten oder *OIds* sortiert ist, wobei dann die *OIds* durch *VOIds* dargestellt werden. Diese Form der Darstellung hat nicht nur den Vorteil der Reduzierung des Speicherverbrauchs, sondern speziell bei der Reproduzierung der originalen Relation ist kein zusätzlicher Aufwand notwendig, da konzeptuell einfach alle benötigten Tail-Spalten nebeneinander gelegt werden und dann schrittweise Tupel um Tupel erzeugt wird.

Neben dem Konzept der BATs werden in MonetDB noch Indexstrukturen genutzt, um den Zugriff zu beschleunigen, darunter binäre Joinindizes (Def. 3.4). Sie werden ebenfalls in BATs gespeichert, wobei Head und Tail vom Typ *OId* sind und auf die „gejointen“ Tupel referenzieren. Um Platz und Zeit bei der Anfrageausführung zu sparen, kann bei einem $1 : N$ Join die 1-Seite unter Umständen als *VOId* dargestellt werden. Für weitere Beschleunigungen während der Anfragebearbeitung wurden Techniken wie Database Cracking, Recycling und Self-organizing Tuple Reconstruction entwickelt [86, 87, 101].

3.2.3 Das Datenmodell PAX

PAX – Partition Attributes Across

Das Datenmodell **PAX** ist eine Mischung aus NSM und DSM [7]. Dabei werden wie im NSM alle Daten einer bestimmten Anzahl von k Tupel auf einer Seite gespeichert, im Gegensatz zum NSM werden allerdings die Daten der einzelnen Tupel nicht hintereinander platziert; stattdessen werden die Tupel wie im DSM bzgl. ihrer Attribute zerlegt und dann attributweise gespeichert. PAX ist somit ein hybrider Ansatz, der bei einer Tupelanzahl von $k = 1$ in das NSM übergeht. Falls k die Anzahl der Tupel in der Relation ist, geht PAX in das DSM über. Der Vorteil des hybriden Ansatz ist die nahe Speicherung der Daten eines einzelnen Tupels und zu gleich eine gewisse örtliche Lokalität der einzelnen Attributwerte eines Attributs.

Beispiel 3.4: Beispiel zum PAX Datenmodell

Abbildung 3.4 zeigt die physische Abbildung mit Hilfe des PAX Datenmodells. Ausgehend von der konzeptuellen Sicht (links), wie in den Beispielen 3.2 und 3.3 wird rechts die physische Sicht dargestellt, bei der immer k Tupel zusammen gespeichert werden.

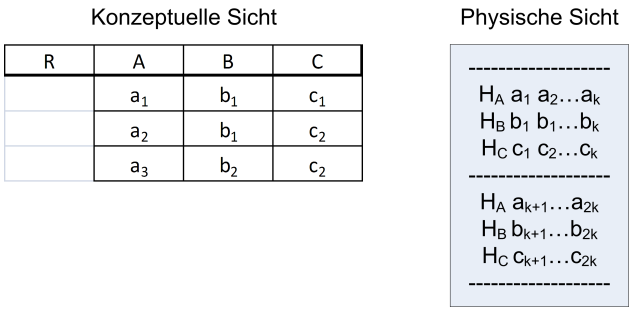


Abbildung 3.4: Beispiel zum PAX Datenmodell

3.2.4 Vergleich der physischen Datenmodelle

Nachdem die drei verschiedenen physischen Datenmodelle NSM, DSM und PAX vorgestellt wurden, sollen sie nun miteinander verglichen und bzgl. der Anforderungen und Verwendbarkeit in dieser Arbeit evaluiert werden. Das physische Datenmodell definiert den (physischen) Zugriff auf auf die Daten und entscheidet somit über die Effizienz des Zugriffs auf Relationen bzw. Tupel. Um das Ziel (s. Abschn. 1.3) der Arbeit die effizient Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen zu verwirklichen, standen die folgenden beiden Eigenschaften des physischen Datenmodells im Vordergrund: (i) Eine hohe Lokalität bei Datenzugriffen und damit Cache-effizient und (ii) müssen sich die Zugriffe einfach und effizient parallelisieren lassen.

Der Fokus dieser Arbeit liegt auf effizienter Anfragebearbeitung im Data-Warehouse-Umfeld (s. Def. 1.1, Def. 1.2 sowie Def. 1.4). Ein Ziel des OLAP-Bereichs ist es, große Mengen an Daten möglichst komprimiert und übersichtlich darzustellen. Dazu werden Anfragen verwendet, die die großen Mengen an vorhandenen Daten selektieren, gruppieren und jeder Gruppe Kennwerte zuweisen. Im Gegensatz dazu werden beim OLTP-Bereich meist einzelne Tupel verarbeitet und verändert. Daraus ergeben sich zwei grundsätzlich unterschiedliche Auswertungsstrategien. Bei OLTP-Anfragen müssen für eine effiziente Anfragebearbeitung die Daten eines einzelnen Tupels möglichst dicht beieinander gespeichert werden, um eine hohe Datenlokalität zu erreichen, zum anderen sollen möglichst wenig Objekte gesperrt werden, um einen hohen Durchsatz (s. Abschn. 1.2.2) zu generieren. Die OLTP-Anfragebearbeitung ist somit tupel-orientiert und damit „horizontal ausgerichtet“. Im OLAP-Bereich werden meist viele Tupel durch die Selektion ausgewählt und anschließend gruppiert; diese Art der Anfragebearbeitung kann als „vertikal“ charakterisiert werden.

Der Unterschied zwischen NSM und DSM wurde bereits in verschiedenen Artikeln untersucht, darunter in [7, 66, 2, 176]. Dabei herrscht grundsätzlicher Einklang darüber, dass das DSM im OLAP-Bereich das geeignetere Datenmodell darstellt. Sattler fasst die Vorteile in [147] wie folgt zusammen. (i) Verarbeitung: Das DSM lässt eine Implementierung von Operatoren auf eine „Cache- und CPU-Pipeline-freundliche“ Art und Weise zu. (ii) Komprimierung: Über das DSM lassen sich Bereiche mit gleichen Attributwerten

einfacher und besser komprimieren. In dieser Arbeit wird dieser Beurteilung gefolgt und daher das spaltenorientierte Datenmodell als Modell der physischen Schicht gewählt.

PAX stellt im Zusammenhang mit Mehrkern-Rechnerarchitekturen eine interessante Variante dar, denn durch die Gruppierung von Mengen von Tupeln auf einer Seite, stellt es eine natürliche Aufteilung für die parallele Abarbeitung dar. PAX wurde bei der Implementierung der Konzepte dieser Arbeit allerdings nicht verwendet, obwohl die Konzepte auch auf PAX anwendbar wären. Der Grund liegt darin, dass mit MonetDB ein echtes, stabiles und vollständiges RDBMS gefunden wurde, auf dem die Konzepte getestet werden konnten und somit qualitativ bessere und auch aussagekräftigere Messungen erwartet werden, als mit einem proprietären, unvollständigem RDBMS, das PAX implementiert.

DSM stellt somit die Basis bzgl. der beiden oben genannten Punkte. Es ist (i) Cache-effizient, und es lässt sich gut parallel darauf arbeiten. Zum einen können mehrere Selektionen parallel abgearbeitet werden, zum anderen kann die Selektion selbst natürlich auch parallelisiert werden. Allerdings müssen für effiziente Anfragebearbeitung auch die Ergebnisse der einzelnen Operatoren wiederum gut parallel verarbeitbar sein. Im nächsten Teil sollen dazu Bit-Strings als Realisierung zur Repräsentation für Tupelmengen vorgestellt werden. Im Kapitel 4 wird dann gezeigt, wie eine effiziente und parallele Anfrageausführung mit Hilfe dieser Bit-Strings aussehen kann.

3.3 Das physische Datenmodell des Frameworks

In diesem Abschnitt soll das physische Datenmodell vorgestellt werden, das Teil des später definierten Framework (s. Abb. 3.5) sein wird. Darauf aufbauend wird das Konzept von globalen Tupelidentifikatoren (gTIDs) eingeführt. gTIDs bzw. gTID-Mengen werden im Laufe dieser Arbeit eine entscheidende Rolle spielen, wenn es um die Abarbeitung von Anfragen geht und darum, wie Ergebnisse bzw. Zwischenergebnisse repräsentiert werden. Im Anschluss werden drei verschiedene Möglichkeiten vorgestellt, das Konzept der globalen TIDs für unterschiedliche Arten von Indizes zu nutzen, die speziell im OLAP-Bereich sinnvolle Erweiterungen zu existierenden Indexarten darstellen. Weiter wird gezeigt, wie gTIDs bei der Modellierung genutzt werden können, um Surrogate (Def. 3.5) zu generieren und welche Vorteile die Nutzung von gTIDs dabei hat. Als eine mögliche Realisierung zur Repräsentation von gTID-Mengen wird das Konzept von Bit-Strings vorgestellt. Den Abschluss stellen Operatoren dar, die auf dem physischen Modell definiert sind und die abstrakten Operatoren aus dem Hardwaremodell Abschnitt 2.3 verwenden.

3.3.1 Eindeutige Identifikatoren für alle Tupel der Datenbank

Das Framework beinhaltet die Verwendung von **globalen Tupelidentifikatoren** (gTIDs), mit denen jedem Tupel der Datenbank durch die Funktion $gTID$ ein eindeutiger Identifikator zugewiesen wird. $gTID$ ist wie folgt definiert:

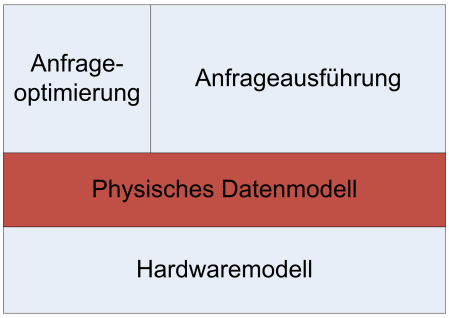


Abbildung 3.5: Framework: Das physische Datenmodell

Definition 3.6 (gTID) Für zwei beliebige Tupel $t_1 \in R_x$ und $t_2 \in R_y$ aus den Relationen R_x und R_y gilt:

$$gTID(t_1) \neq gTID(t_2) \Leftrightarrow t_1 \neq t_2$$ □

Im Gegensatz zu Tupelidentifikatoren (TIDs), die auf Relationsebene definiert sind und somit zwei Tupel $t_1 \in R_x$ und $t_2 \in R_y$ in den Relationen R_x und R_y ($R_x \neq R_y$) existieren können, so dass $TID(t_1) = TID(t_2)$ ist, erlaubt die Verwendung von globalen Tupelidentifikatoren nicht nur Mengen von Tupeln bzgl. einer einzelnen Relation, sondern auch relationsübergreifend zu verwalten.

Weiter definiert das Framework explizit die **Extension** der Relation R als Menge von gTIDs über die Funktion Ext und die Menge der gTIDs der Relation R bzgl. einer beliebigen gTID-Menge T über die Funktion $Proj$.

Definition 3.7 (Funktionen *Ext* und *Proj*) Für ein Tupel t, eine Relation R und eine gTID-Menge T gilt:

$$\begin{aligned} gTID(t) \in Ext(R) \Leftrightarrow & t \in R \\ Proj(R,T) = & Ext(R) \cap T \end{aligned}$$ □

Im Folgenden soll die Verwendungsmöglichkeit von globalen Tupelidentifikatoren an drei Indexarten vorgestellt werden.

3.3.1.1 Der Domänenindex

Eine mögliche Verwendung der globalen Tupelidentifikatoren ist die Nutzung von **Domänenindizes** (Def. 3.3) [131, 146].

Definition 3.8 (*DomIdx*) Der Domänenindex $DomIdx(D)$ ist eine Menge von Paaren, bestehend aus einem Attributwert aus der Domäne D und einer Menge von gTIDs. Für eine Menge von Relationen $\mathcal{R} = \{R_1, ..., R_n\}$ ist der Domänenindex bzgl. der Domäne D wie folgt definiert:

$$DomIdx(D) = \{[d,T] | \forall d \in D \wedge T = \bigcup_{k=1...n} domGTID(R_k, D, d)\}$$ □

Dabei gibt die Funktion $domGTID(R_k, D, d)$ die Menge aller gTIDs zurück, für die die korrespondierenden Tupel aus R_k bzgl. der Domäne D den Wert d besitzen.

$$\begin{aligned} domGTID(R_k, D, d) =& \{gTID(t) | \forall t \in R_k \\ & \wedge \exists A \in sch(R_k) \wedge dom(A) = D \wedge \mu_t(A) = d\} \end{aligned}$$

Um den Domänenindex sinnvoll einzusetzen, werden noch zwei weitere Funktionen definiert, die es ermöglichen den Index bzgl. eines Attributwertes d anzufragen. Die Funktion $GetDomIdx(D, d) = T$ liefert die gTID-Menge T für den Indexeintrag mit dem Wertepaar $[d, T] \in DomIndex(D)$ aus dem Domänenindex für D zurück und $GetDomIdxRel(D, d, R)$ beschränkt die Menge der gTIDs auf die Relation R.

$$\begin{aligned} GetDomIdx(D, d) =& T \Leftrightarrow [d, T] \in DomIdx(D) \\ GetDomIdxRel(D, d, R) =& Proj(R, GetDomIdx(D, d)) \end{aligned}$$

Mit Hilfe der Funktion $GetDomIdxRel(D, d, R)$ ist es möglich, einen Index auf einer einzelnen Relation R zu simulieren. Der Vorteil bei der Verwendung des Domänenindexes liegt in der einmaligen Speicherung der Attributwerte in der Indexstruktur, so dass eine höhere Lokalität sowie eine Reduktion des Speicherverbrauchs ermöglicht wird. Diese Vorteile müssen mit der zusätzlich notwendigen Schnittmengenbildung in der Funktion $Proj$ bezahlt werden. Im Abschnitt 3.3.2 wird gezeigt, wie mit Hilfe von Bit-Strings, als Realisierung von gTID-Mengen, diese Operation effizient und auch parallel abgearbeitet werden kann. Eine weitere Möglichkeit der Nutzung von Domänenindizes ist bei der Equijoin- bzw. Naturaljoin-Berechnung (s. Abschn. 3.1) unter Verwendung von Joinindizes.

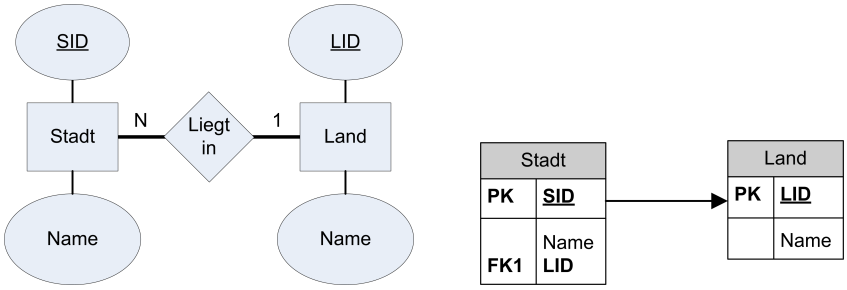
3.3.1.2 Der Joinindex

Der Joinindex [131] ist eine spezielle Form eines Indexes, bei dem Beziehungen zwischen mehreren Relationen vorberechnet und abgespeichert werden (Def. 3.4). Nachfolgend soll am Beispiel einer 1:N-Beziehung der Aufbau und die Verwendung eines Joinindexes gezeigt werden.

Beispiel 3.5: Aufbau und Verwendung eines Joinindexes

```
Q1: SELECT Land.Name FROM STADT, LAND
    WHERE STADT.Name = 'Berlin'
      AND STADT.LID = Land.LID
Q2: SELECT Count(*) FROM STADT, LAND
    WHERE LAND.Name = 'Germany'
      AND STADT.LID = Land.LID
```

Gegeben seien zwei Entitäten *Stadt* und *Land*, so dass jede Stadt in genau einem Land liegt und ein Land mehrere Städte haben kann (s. Abb. 3.6(a)). Bei der Überführung des



(a) ER-Modell (b) Relationen-Modell

Stadt

TID	SID	Name	LID
0	BER	Berlin	D
1	HAM	Hamburg	D
2	NY	New York	US
3	LON	London	GB

Land

TID	LID	Name
4	D	Germany
5	US	USA
6	GB	Great Britan

Joinindex

TID	TID
0	4
1	4
2	5
3	6

(c) Extension

Abbildung 3.6: Modellierung Stadt,Land und „Liegt in“

ER-Modells in das Relationenmodell existieren mehrere Möglichkeiten. Im Beispiel bilden beide Entitäten jeweils eine Relation und die Beziehung wird innerhalb der Relation Stadt modelliert (s. Abb. 3.6(b)). Abbildung 3.6(c) gibt dazu eine mögliche Ausprägung und den zugehörige Joinindex, wie er z.B. in MonetDB verwendet wird (links). Seine Verwendung soll kurz an den beiden Beispielanfragen Q_1 und Q_2 vorgestellt werden.

Anfrage Q_1 fragt nach dem Land, in dem die Stadt „Berlin“ liegt. Ohne Verwendung des Joinindexes müsste zuerst das passende Tupel in *Stadt* ($TID = 0$) gefunden werden und danach das passende Tupel in *Land* ($TID = 4$). Durch den Joinindex ist diese Information bereits vorberechnet. Wir können in diesem Fall von der Tupelposition von Tupel $TID = 0$ direkt auf Joinindexposition schließen und damit das entsprechende Jointupel in *Land* lesen.

Die Anfrage Q_2 dagegen fragt nach der Anzahl aller Städte im Land mit dem Namen „Germany“. Bei der Beantwortung der Anfrage müsste man entweder die Relation *Stadt* oder den Joinindex komplett lesen. Da beide gleich viele Einträge haben, kann der Joinindex in diesem Fall den Aufwand nicht reduzieren und ist in diesem Fall nur in einer Richtung effizient.

Eine andere Möglichkeit der Konstruktion eines Joinindexes ist es, für jedes Tupel aus der Relation *Land* die korrespondierende Menge von TIDs bzgl. der Relation *Stadt* zu vermerken [29, 103]. Dieser Index funktioniert wiederum nur für die Richtung von Land nach Stadt. Sollen beide Richtungen effizient bearbeitet werden, sind zwei Indizes notwendig und führt zu einer Erhöhung der Kosten bei Veränderungsoperationen, sowie des Speicherbedarfs. Abhilfe kann hier ein Domänenindex auf LID schaffen. Der Domä-

nenindex stellt in diesem Zusammenhang die spezielle Eigenschaft der Naturaljoin dar und funktioniert wie folgt.

DomIdx(LID)

LID	gTIDs
D	{0,1,4}
US	{2,5}
GB	{3,6}

Abbildung 3.7: Domänenindex auf LID

Abbildung 3.7 zeigt die Instantiierung des Domänenindexes $DomIdx(LID)$. Bei der Beantwortung der ersten Anfrage Q_1, sucht man zuerst in *Stadt* nach „Berlin“ und findet das entsprechende Tupel mit dem Eintrag „D“ in LID, damit ergibt sich aus dem Index mit Hilfe der Funltion $GetDomIdxRel(LID, \text{„D“}, LAND) = \{4\}$. Es ist somit die $gTID$ des Joinpartners in *Land* gefunden worden.

Für die zweite Anfrage Q_2 wird zuerst nachdem entsprechende Land mit dem Namen „Germany“ gesucht, um seinen LID Attributwert nutzen zu können. Der Index gibt die Menge $\{0, 1\}$ für $GetDomIdxRel(LID, \text{„D“}, STADT)$ zurück, damit kann direkt über die Größe dieser Menge die Anfrage beantwortet werden. Mit Hilfe des Domänenindexes auf dem Joinattribut können somit beide Join-Richtungen durch einen einzigen Index effizient bearbeitet werden.

3.3.1.3 Der Bereichsindex

Im OLAP-Bereich werden häufig nur bestimmte Bereiche der Attributwerte (z.B. Datum oder Zeit) ausgewählt. Die Verwendung eines normalen Indexes ist dabei oft nicht sinnvoll, besonders in Fällen bei denen der Bereich genügend groß ist, so dass sich sehr viele verschiedene Attributwerte qualifizieren. Genauer gesagt sind bei n vielen unterschiedlichen Attributwerten innerhalb des angefragten Bereichs n Indexzugriffe notwendig und $n - 1$ Vereinigungsoperationen der n TID-Mengen. Eine Möglichkeit diesen Aufwand zu minimieren und die Zugriffe auf den Index auf eine konstante Anzahl von maximal zwei, sowie eine einzige Mengenoperation zu reduzieren, wird mit einem Bereichsindex erreicht.

Definition 3.9 (RangeIdx) Ein **Bereichsindex** $RangeIdx(R, A)$ auf der Relation R und dem Attribut A aus R ist eine Menge von Paaren aus einem Attributwert a und einer TID-Menge T, dabei muss auf A eine totale Ordnung definiert sein. Dann gilt:

$$RangeIdx(R, A) = \{[a, RangeTIDs(R, A, a)] | \forall a \in dom(A)\}$$

wobei RangeTIDs(R,A,a) definiert ist als

$$RangeTIDs(R, A, a) = \{gTID(t) | \forall t \in R \wedge \mu_t(A) \leq a\}$$ □

Für die Suche nach Tupeln aus R, die innerhalb des linksoffenen Intervalls $(a_l, a_h]$ liegen, ist die Funktion $GetRangeIdx(R, A, a_l, a_h)$ definiert mit:

$$\begin{aligned} GetRangeIdx(R, A, a_l, a_h) =& TID_h \setminus TID_l \\ & \Leftrightarrow \\ \{[a_l, TID_l], [a_h, TID_h]\} \subseteq& RangeIdx(R, A) \end{aligned}$$

Die Definition von $RangeIdx(R, A)$ ist über alle möglichen Domänenwerte definiert. Dies führt zu einer unnötigen Ressourcenverschwendung, so dass bei einer Implementierung nur Attributwerte indexiert werden, die auch in der Datenbank gespeichert sind. Für die Suche mit $GetRangeIdx$ muss dann die Definition entsprechend angepasst werden.

Um Punktanfragen eines Attributwertes a über einen Bereichsindex $RangeIdx(R, A)$ zu beantworten, wird das linksoffene Intervall $(a_l, a]$ mit $a_l \prec a$ ausgewertet, dabei ist a_l der direkte Vorgänger von a, der einen Eintrag im Index besitzt. Für Punktanfragen erhöhen sich somit die Kosten um die Differenzbildung und das Lesen eines zusätzlichen Indexeintrags, dafür können Intervalle von beliebiger Länge mit konstantem Aufwand beantwortet werden. Ähnlich zur Definition des Domänenindexes (Def. 3.8) kann auch der Bereichsindex auf Domänenebene, anstatt auf Relationsebene, definiert werden.

3.3.1.4 Anfragebearbeitung bei Negation

Eine besondere Herausforderung stellt die Negation dar, wie sie in der Anfrage in Beispiel 3.6 verwendet wird.

Beispiel 3.6: Anfrage mit Negation

```
SELECT Name FROM STADT WHERE Name <> 'Berlin'
```

Mit der obigen Anfrage sollen die Namen aller Städte außer „Berlin“ ausgegeben werden. Die Verwendung eines Indexes auf dem Attribut $Name$ kann hier zu keiner Verbesserung der Anfragebearbeitungszeit führen, daher wird bei solchen Anfragen die gesamte Relation gelesen, jedes Tupel auf den entsprechenden Attributwert geprüft und dann weiterverarbeitet.

Die Ext Funktion (Def. 3.7) kann hier zu einer starken Leistungsverbesserung beitragen. Die Idee dabei ist, einfach von der gesamten Menge an TIDs der Relation diejenigen abzuziehen, die nicht in das Ergebnis gehören. Sei die Funktion $GetIndex(R, A, a)$ definiert als die Menge an TIDs, bei denen das Tupel in R den Wert a im Attribut A hat. Die Funktion $GetIndexNot(R, A, a)$ berechnet die Menge an TIDs bei denen der Wert des Attributs A unterschiedlich von a ist (und ungleich von `NULL`).

Definition 3.10 (IndexNot)

$$\begin{aligned} GetIndex(R, A, a) =& \{gTID(t) | \forall t \in R \wedge \mu_t(A) = a\} \\ GetIndexNot(R, A, a) =& Ext(R) \setminus GetIndex(R, A, a) \\ & \setminus GetIndex(R, A, \texttt{NULL}) \end{aligned}$$ □

Im Beispiel 3.6 werden mit $GetIndexNot(STADT, Name, 'Berlin')$ die entsprechenden Tupel bestimmt und ihre Namenswerte ausgegeben. Der Vorteil dieser Art der Abarbeitung liegt in der Nutzung des Indexes auf den Nameswerten, so dass die Tupel ohne einen teuren Zeichenkettenvergleich auf „Berlin“ direkt ausgegeben werden können.

3.3.1.5 Globale Tupelidentifikatoren als Surrogate

Eine andere wichtige Art der Nutzung von gTIDs (Def. 3.6) ist ihre Verwendung als Surrogate (Def. 3.5). Sie werden oft der Modellierung eingefügt, haben aber keinerlei Bedeutung, außer bei der Vereinfachung der Joinberechnung auf Primär- und Fremdschlüsselbeziehungen, da anstatt eines komplizierten Textvergleiches oder mehrerer Datenvergleiche, nur ein einziger nummerischer Vergleich ausgeführt werden muss.

Das Framework stellt dem Datenbanknutzer die globalen TIDs als Eigenschaft von jedem Tupel zur Verfügung, er darf dabei zwar nicht direkt auf den Wert der jeweiligen gTID zugreifen, aber es soll möglich sein, gTIDs als Surrogate zu nutzen. Im Framework sollen alle Fremdschlüsselbeziehungen auf Basis der gTIDs umgesetzt werden. Dazu kommt noch ein spezieller Domänen-Joinindex, der auf der Domäne der TID definiert ist. Über diesen Index kann zum einen, wie in den Beispielen zuvor beschrieben, die Joinbearbeitung effizienter gestaltet werden, zum anderen können sehr viel einfacher referenzielle Integritäten sichergestellt werden. Dies soll an einem Beispiel vorgestellt werden.

Beispiel 3.7: gTIDs anstatt Surrogate

Aufbauend auf Beispiel 3.5 werden die beiden Relationen *Stadt* und *Land* verwendet, nur dass die Primär- und Fremdschlüssel durch gTIDs ersetzt sind. *Land* besteht somit aus den Attributen TID und $Name$. *Stadt* besteht aus TID, $Name$ und $LTID$, wobei $LTID$ von der gleichen Domäne ist wie TID und auf das entsprechende Tupel in *Land* referenziert.

Es soll das Land „Great Britan“ aus der Relation *Land* gelöscht werden. Mit Hilfe des Domänen-Joinindex $DomIdx(TID)$ kann die Menge $T = \{3, 6\}$ aller Tupel berechnet werden die mit dem Tupel „Great Britan“ ($TID = 6$) in Verbindung stehen und daraus wird erkannt, dass das zu löschende Tupel noch referenziert wird.

Die Verwendung von gTIDs anstelle von benutzerdefinierten Surrogaten hat folgende Vorteile:

- Beschleunigung der Anfragebearbeitung bei Joins auf Fremdschlüsselbeziehungen durch den Domänen-Joinindex.

- Reduzierung des Speicherverbrauchs, da die benutzerdefinierten Surrogate nicht mehr gespeichert werden müssen.
- Vereinfachung bei der Modellierung des relationalen Modells, da implizite Identifikatoren genutzt werden können und somit nicht selbst modelliert werden müssen.
- Vereinfachung bzw. Verbesserung des Sicherstellens von referenzieller Integrität.

3.3.2 Bit-Strings als Repräsentation für TID-Mengen

Ein wichtiger Punkt im Zusammenhang mit TID-Mengen ist (i) die physische Darstellung dieser Mengen und (ii) ihre effiziente Verarbeitung. 1987 hat Patrick O'Neil in seiner Arbeit die Verwendung von Bit-Strings oder Bitmaps vorgeschlagen [129].

Definition 3.11 (Bit-String) Ein **Bit-String** repräsentiert eine Menge von TIDs als eine binäre Folge von Nullen und Einsen. Die TIDs werden dabei durch das Bit an der Position ihres TID Werts repräsentiert. Eine Eins bedeutet, die TID ist in der Menge enthalten; eine Null bedeutet, sie ist nicht enthalten. □

Die Verwendung von Bit-Strings bzw. Bit-String-Indizes im Rahmen der Anfragebearbeitung im Data-Warehouse-Bereich wurde von O'Neil und Graefe [130] vorgeschlagen und ist seitdem stark verbreitet [154, 131, 31, 32, 172, 144, 12, 171, 13, 103]. Dabei werden verschiedene Prädikate auf einer gegebenen Relation mit Hilfe von Bit-String-Indizes beantwortet. Die Ergebnisse können dann durch bitweise Und- und Oder-Operationen verknüpft werden und stellen dabei den Schnitt bzw. die Vereinigung dar.

Beispiel 3.8: Anfrageverarbeitung mit Bit-Strings

```
SELECT A1, COUNT(A2) FROM R
  WHERE A3 = 4711 AND A4 = 'Auto'
  GROUP BY A1
```

Die Anfrage in Beispiel 3.8 kann mit Hilfe eines Bit-String-Indizes, unter Verwendung der Beispielimplementierung in Beispiel 3.9 wie folgt beantwortet werden. Zuerst wird der Bit-String bs_{4711} für $A_3 = 4711$ durch einen Indexzugriff (Def. 3.10) nachgeschlagen, ebenso der Bit-String bs_{Auto} für $A_4 =$ "Auto" (Zeile 3,4). Danach kann die Schnittmenge der potentiellen Tupel durch das bitweise *Und* der beiden Bit-Strings $bs_{where} = bs_{4711} \wedge bs_{Auto}$ (Zeile 6) gebildet werden. Anschließend wird für jeden Attributwert von A_1 der jeweilige Bit-String nachgeschlagen (Zeile 8,9) und mit bs_{where} wiederum durch bitweises *Und* verknüpft (Zeile 10), die Anzahl der Einsen gezählt (Zeile 11) und entsprechend ausgegeben (Zeile 15).

Beispiel 3.9: Implementierung der Anfrage Bsp. 3.8

```
1 list<pair<String,int>> Anfrage(){
2     list<pair<String,int>> rs;
3     BitString bs4711 = GetIndex(R,A3,4711);
4     BitString bsAuto = GetIndex(R,A4,"Auto");
5
```

```
6     BitString bsWhere = bs4711 & bsAuto;
7
8     foreach(pair<String,BitString> a1 :
9         GetValuesAndBitStrings(R,A1)){
10        BitString result = a1.bs & bsWhere;
11        int count = GetBitCount(result);
12        if(count != 0)
13            rs.append(pair<String,int>(a1.string,count));
14    }
15    return rs;
16 }
```

Im Data-Warehouse-Bereich werden außerdem seit langem Bit-String-Joinindizes verwendet [103, 29], bei denen zuerst die Dimensionstabellen durch Prädikate gefiltert werden, ehe anschließend ein Join bzw. eine Selektion auf der Faktentabelle mit Hilfe der Bit-String-Joinindizes erfolgt.

In dieser Arbeit sollen Bit-Strings im gesamten Prozess der Anfrageausführung genutzt werden. Es sollen daher nicht nur die Filterbedingungen mit Hilfe von Bit-Strings beantwortet werden, sondern auch Joins und Gruppierungen. Dabei werden Details zur Ausführung im Abschnitt 4.3 vorgestellt. Der Grund für die Verwendung von Bit-Strings liegt in den folgenden Punkten:

- Bit-Strings stellen eine Datenstruktur dar, mit der sehr einfach gearbeitet werden kann (einfache Und-, Oder- und Negationsoperationen).
- Da Bit-Strings eine lineare Anordnung im Speicher haben, ist der Zugriff und die Verarbeitung sehr effizient, speziell durch prozessorinternes Prefetching.
- Große Bit-Strings lassen sich einfach zerlegen und damit parallelisieren.

3.3.3 Operatoren des Datenmodells

In diesem Teilabschnitt werden Operatoren vorgestellt, die auf dem physischen Datenmodell agieren. Neben den Standardoperatoren des DSM, wie z.B. Operatoren, die einen Attributwert eines Attributes eines Tupel lesen und den zuvor vorgestellten Indexoperatoren, kommen noch Operatoren auf Bit-String-Ebene hinzu. Dazu sollen im Folgenden zuerst, anhand des bitweisen *Und*-Operators, Operatoren vorgestellt werden, die nur Bit-Strings als Ein- und Ausgabe verarbeiten. Danach werden exemplarisch weitere Operatoren für die Verbindung von Bit-String mit dem DSM präsentiert. Bei der Darstellung wird gezeigt, wie die abstrakten Operatoren aus Abschnitt 2.3 angewendet werden. Die eigentliche parallele Ausführung mit Hilfe des Hardwaremodells folgt dann im Abschnitt 4.3 des nächsten Kapitels.

3.3.3.1 Operatoren auf Bit-Strings

Als Ausprägungen von Operatoren bei denen sowohl die Eingabe als auch die Ausgabe Bit-Strings darstellen, existieren das bitweise *Und* und *Oder*, die Negation und die Sub-

traktion. Im Anschluss soll der *Und*-Operator vorgestellt werden. Während die Negation ein unärer Operator ist, und die Subtraktion binär arbeitet, sind *Und* und *Oder* N-näre Operatoren. Zur Vereinfachung wird aber im Beispiel ein binäres *Und* erklärt (Bsp. 3.10).

Der Beispieloperator (Bsp. 3.10 Zeile 27 bis 40) führt ab der Position *offset* und über eine Länge von *length*, eine bitweise *Und*-Operation zweier Bit-Strings aus. Dabei verwendet die Implementierung den abstrakten Bibliotheksoperator *ABSTRACT_AND*; die eigentliche Bitverknüpfungsoperation ist hardwarearchitekturabhängig implementiert und wird durch eine Bibliothek eingebunden. Die Zeilen 8 bis 25 zeigen eine mögliche Implementierung durch SIMD-Operationen. Wichtig ist, dass bei der Ausführung die auszuführende Funktion bereits bekannt ist und nicht durch das späte oder dynamische Binden erfolgt. Durch den anderen abstrakten Bibliotheksoperator *ABSTRACT_PREFETCH* werden die Daten architekturabhängig vorzeitig in den Cache geladen. Wie man sieht, ist der Operator selbst nicht parallel, er stellt somit die kleinste nicht parallelisierbare[1] Einheit dar. Bei der parallelen Ausführung wird dann ein großer Bereich eines Bit-Strings rekursiv partitioniert (s. Bsp. 2.9) bis die gewünschte Granularität erreicht ist. Auf diesem Teilstück wird dann jeweils der Operator *AndOperator* ausgeführt. Damit können verschiedene Tasks parallel auf unterschiedlichen Bereichen der Bit-Strings agieren.

Beispiel 3.10: Der bitweise *Und*-Operator

```
1  class ABSTRACT_AND
2  {
3      void OP(BitString& left, BitString& right,
4          BitString& result);
5      size_t LEN();
6  };
7
8  class SIMD_AND : public ABSTRACT_AND
9  {
10     void OP(BitString& left, BitString& right,
11         BitString& result)
12     {
13         for(int i=0; i<8; ++i)
14         {
15             __m128i left128  = _mm_stream_load_si128(
16                 ((__m128i*) left) + i);
17             __m128i right128 = _mm_stream_load_si128(
18                 ((__m128i*) right) + i);
19             __m128i rc128 = _mm_and_si128(left128,right128);
20             _mm_stream_store_si128(((__m128i*) result) + i,
21                 rc128);
22         }
23     }
24     size_t LEN(){return 128;}
25 }
26
```

[1] außer durch SIMD

```
27 void AndOperator(BitString& left, BitString& right,
28     BitString& result, size_t offset, size_t length)
29 {
30     size_t skip = ABSTRACT_AND.LEN();
31     size_t end = offset + length;
32     ABSTRACT_PREFETCH(offset,end,skip,left,right,result);
33     while(offset < end)
34     {
35         ABSTRACT_AND.OP(left[offset],right[offset],
36             result[offset]);
37         offset += skip;
38         ABSTRACT_PREFETCH(offset,end,skip,left,right,result);
39     }
40 }
```

3.3.3.2 Operatoren des physischen Datenmodells

Neben den Bit-String-Operatoren existieren noch Operatoren, die entweder Bit-Strings aus den Daten erzeugen oder mit Hilfe von Bit-Strings Daten verarbeiten. Beides soll anhand von vier Operatoren vorgestellt werden. Der erste Operator dient dabei dazu, einen Bit-String bzgl. eines Attributes einer Relation zu erstellen. Er wird verwendet, wenn kein Index für das jeweilige Attribut vorhanden ist, aber trotzdem eine Selektion in der Anfrage bzgl. dieses Attributs definiert wurde und die Darstellung des Ergebnisses der Selektion durch einen Bit-String für die Weiterverarbeitung notwendig ist. Der Operator *GetBitStringBySelection* erstellt somit den entsprechenden Bit-String zur Laufzeit („on-the-fly“).

Beispiel 3.11: Der Operator GetBitStringBySelection

```
void GetBitStringBySelection(Column& column, Prädikat& p,
    BitString& result, size_t offset, size_t length){
    size_t skip = ABSTRACT_SEL.LEN();
    size_t end = offset + length;
    while(offset < end)  {
        ABSTRACT_SEL.OP(c[offset],p,result[offset]);
        offset += skip;
    }
}
```

Der zweite Operator *AggSumByBitString* führt eine Aggregation, in diesem Fall eine Addition auf einem Attribut einer Relation aus, bei dem nur die im Bit-String markierten Tupel aggregiert werden.

Beispiel 3.12: Der Operator AggSumByBitString

```
int AggSumByBitString(Column& column, BitString& bs,
    size_t offset, size_t length){
    int sum = 0;
    size_t skip = ABSTRACT_AGG_SUM.LEN();
```

```
    size_t end = offset + length;
    while(offset < end)  {
        sum += ABSTRACT_AGG_SUM.OP(column[offset],
            bitstring[offset]);
        offset += skip;
    }
    return sum;
}
```

Bei beiden Operatoren wurden zusätzliche Dinge – beispielsweise Prefetching von Daten – zur vereinfachten Darstellung weggelassen. Die Herzstücke stellen jeweils die abstrakten Operatoren *ABSTRACT_SEL* und *ABSTRACT_AGG_SUM* dar. Sie stellen Funktionen der Klasse von statischen Laufzeitoperatoren (s. Abschn. 2.3.2.2) dar, denn erst zur Plangenerierungszeit wird die beste Implementierung für den jeweiligen Typ und die jeweiligen Bedingungen gewählt.

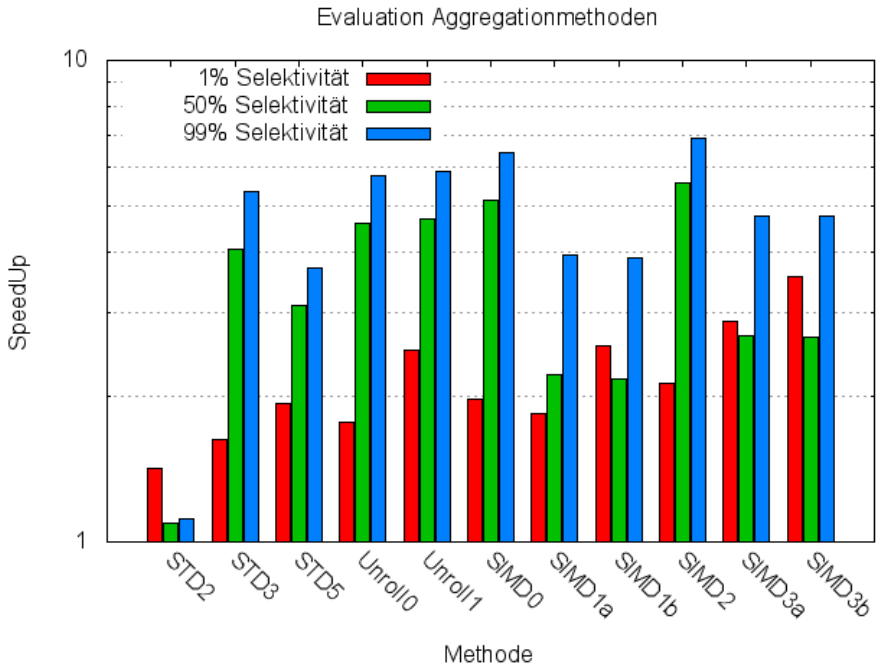
Für den Operator *ABSTRACT_AGG_SUM* wäre eine Implementierungsmöglichkeit, das Testen eines Bits im Bit-String und in Abhängigkeit darauf die Addition auszuführen. Eine andere Möglichkeit ist die Nutzung der Eigenschaften von 0 und 1 bei der Multiplikation, bei der wird zuerst das jeweilige Bit mit dem zu addierenden Wert multipliziert und anschließend wird addiert. Der Vorteil ist, dass der Prozessor keine Bedingung in seiner Pipeline hat und es somit auch nicht zum Pipelineflush kommt, wenn eine Fehlannahme bei der Sprungvorhersage gemacht wurde (s. Abschn. 2.1.2.2). Der Nachteil dieser Methode ist, dass jeder Wert addiert wird und damit auch geladen wird, was bei sehr geringer Selektivität, also vielen Null Additionen, zu überflüssigen Speichertransfers führt.

Evaluation des Aggregationsoperators

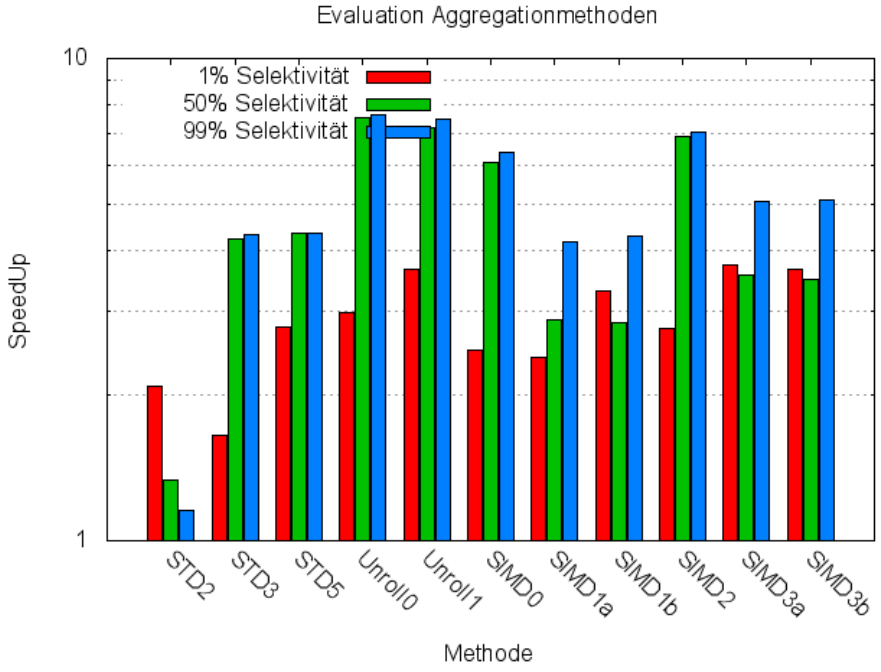
Im Folgenden werden verschieden Implementierungen für die Aggregierung mit Hilfe eines Bit-Strings evaluiert; dabei wurde auch der Einfluss der Selektivität bzgl. des Bit-Strings untersucht. Abbildung 3.8 stellt die Ergebnisse der Messungen dar (Ordinatenachse logarithmisch aufgetragen). Der verwendete Bit-String wurde bzgl. einer bestimmten Selektivität (0,01, 0,5 und 0,99) gleich verteilt initialisiert. Die simulierte Relation hatte 320.000 Tupel, damit war der Bit-String etwa 9 KByte lang. Die Abbildung 3.8(a) zeigt die Ergebnisse unter Verwendung des Microsoft-Compilers, die Abbildung 3.8(b) bei Verwendung des Intel-Compilers. Bei der Evaluierung wurden folgende Implementierungsmöglichkeiten getestet:

STD0 Ist eine Standardimplementierung mit Hilfe von zwei geschachtelten FOR-Schleifen und einem IF-Statement zum Prüfen, ob das entsprechende Tupel selektiert wurde oder nicht. Diese Implementierung stellt den Bezugspunkt (*SpeedUp=1*, Def. 1.8) dar, gegen die alle anderen Methoden evaluiert wurden.

STD2 Wie bei der Standardimplementierung wird in zwei geschachtelten Schleifen vorgegangen, allerdings wird die innere Schleife verlassen, sobald das aktuelle Bit-



(a) Microsoft Compiler



(b) Intel Compiler

Abbildung 3.8: Evaluierung verschiedener Aggregationmethoden

String-Segment (je 32Bit) keine Einsen mehr aufweist (basierend auf bitweisen rechtsschiebe Operationen).

STD3 / STD5 sind äquivalent zu STD0 bzw. STD2 außer das anstatt der IF-Anweisung, die Eigenschaften von Null (absorbierendes Element) und Eins (neutrales Element) verwendet werden, um den bedingten Sprung zu eliminieren.

UNROLL0 / UNROLL1 sind äquivalent zu STD3 bzw. STD5 bis auf das die innere Schleife durch Loop-Unrolling zusätzlich eliminiert wurde.

SIMD0 Die Implementierung von SIMD0 ist in Beispiel 3.13 dargestellt. Man kann sehen, dass analog zu STD0 zwei geschachtelte Schleifen verwendetet werden. Allerdings werden SIMD (Def. 2.8) Befehle verwendet und damit immer vier aufeinander folgende Tupel gleichzeitig verarbeitet.

SIMD1a ist wie SIMD0, jedoch wird vor jedem inneren Schleifendurchlauf getestet, ob überhaupt eines der vier Tupel selektiert wurde.

SIMD1b wie bei SIMD1a wird für jeden inneren Schleifendurchlauf seine Notwendigkeit getestet. Zusätzlich wird aber auch noch die gesamte innere Schleife getestet, ob mindestens eines der nächsten 32 Tupel selektiert wurde.

SIMD2, SIMD3a und SIMD3b sind die bzgl. der inneren Schleife „ausgerollten“ Versionen von SIMD0, SIMD1a und SIMD1b.

Beispiel 3.13: Aggregation mittels SIMD [123]

```
1  uint64_t Agg_SIMD0(uint32_t* column, uint32_t* bs, size_t length){
2      __m128i maskAnd = _mm_set_epi32(8,4,2,1);
3      __m128i zero = _mm_setzero_si128();
4      __m128i sum = _mm_setzero_si128();
5
6      for(size_t i=0;i<length;++i,++bs){
7          __m128i bs128 = _mm_set1_epi32(*bs);
8          for(uint32_t c=0;c<8;++c){
9              __m128i mask128 = _mm_and_si128(bs128,maskAnd);
10             mask128 = _mm_cmpeq_epi32(mask128,maskAnd);
11             bs128 = _mm_srli_epi32(bs128,4);
12
13             __m128i valc = _mm_stream_load_si128(
14                 (__m128i*) column);
15             column += 4;
16             __m128i val = _mm_castps_si128(_mm_blendv_ps(
17                 _mm_castsi128_ps(zero),_mm_castsi128_ps(valc),
18                 _mm_castsi128_ps(mask128)));
19
20             __m128i tmp = _mm_castps_si128(_mm_blend_ps(
21                 _mm_castsi128_ps(zero),_mm_castsi128_ps(val),5));
22             sum = _mm_add_epi64(sum,tmp);
```

```
23
24              val = _mm_srli_si128(val,4);
25              __m128i tmp1 = _mm_castps_si128(_mm_blend_ps(
26                  _mm_castsi128_ps(zero),_mm_castsi128_ps(val),5));
27              sum = _mm_add_epi64(sum,tmp1);
28      } }
29      uint64_t rc = sum.m128i_u64[0] + sum.m128i_u64[1];
30      return rc;
31  }
```

Aus der Abb. 3.8 ergibt sich, dass im Fall der Selektivität von 50% bzw. 99% ein maximaler SpeedUp von sieben gegenüber der Standardimplementierung erreicht werden konnte. Ebenfalls ist der Einfluss der Prüfung zu sehen, ob es sich um ein selektiertes Tupel handelt (Sprung von STD0/ STD2 nach STD3/STD5). Das **Loop-Unrolling**, also die Ausprogrammierung einer Schleife [79], erreicht eine ähnliche bzw. bessere Leistung wie die SIMD-Varianten, da zum einen der Compiler durch automatische Vektorisierung (Def. 2.30) und zum anderen die Reduzierung von Bedingtensprüngen (Faktor 32) eine bessere Auslastung des Pipelinings (s. Abschn. 2.1.2.2) ermöglichen. Es existiert somit nicht nur eine einzige „beste“ Implementierung; aufgrund dieser Abhängigkeit von der Selektivität werden die Operatoren als **statische** Laufzeitoperatoren (s. Abschn. 2.3.2.2) betrachtet. Es ist aber ebenso möglich die Entscheidung auf die Laufzeit zu verlagern, um mittels des Bit-String die exakte Selektivität zu bestimmen und nicht wie bei der statischen Variante Statistiken zu verwenden.

3.3.3.3 Die Operatoren *Group* und *Cross*

In Abschnitt 4.3.2.1 werden für die Anfrageausführung noch zwei weitere Operatoren verwendet: *Group* und *Cross*.

Der *Group*-Operator funktioniert wie die normale Gruppierung. Der Operator bildet bezüglich eines gegebenen Attributes alle Gruppen unterschiedlicher Attributwerte und repräsentiert die passenden Tupel in jeweils einem Bit-String pro Gruppe. Die Ausgabe des *Group*-Operators ist eine Menge von Bit-Strings, die jeweils eine Gruppe repräsentieren.

Die Funktionsweise des *Cross* Operators ist in Beispiel 3.14 dargestellt. Er bildet das kartesische Produkt zweier Bit-String-Mengen durch Schnittmengenbildung.

Beispiel 3.14: Der Cross Operator

```
void CrossOperator(SetBitStrings& s1,
   SetBitStrings& s2 ,SetBitString& result){
   foreach(BitString u: s1){
      foreach(BitString v: s2){
         result.append(ABSTRACT_AND.OP(u,v));
}  } }
```

3.4 Indizes auf SIMD-Rechnerarchitekturen

Im letzten Abschnitt des dritten Kapitels stehen Indizes (Def. 3.2) im Mittelpunkt. Für einen Index existieren verschiedene Möglichkeiten die Schlüssel und zugehörigen Tupel bzw. Tupelreferenzen zu organisieren. Neben dem indexsequentiellen Zugriff, existieren noch Zugriffe über B,B^+,B^*-Bäume, über Hashingtabellen, und viele weitere Indexstrukturen [103]. Rao et al. untersuchen in [139, 140], wie die Suche in B-Bäumen Cache-effizienter gestaltet werden kann. Dazu ordnen sie die Daten so an, dass möglichst viele Cache-Hits (Def. 2.3) produziert werden. Diese Datenstrukturen bzw. Algorithmen nennt man Cache-bewusst (engl. Cache-Aware, Cache-Oblivious) [70].

Eine andere Art der Leistungssteigerung ist die Parallelisierung der Suche selbst. Mit Ausnahme des Hashings wird bei der Suche meist der Suchbereich mit Hilfe von einer binären Suche immer weiter verkleinert bis der jeweilige Schlüssel gefunden wurde. Das bedeutet bei der Suche ist die Komplexität $O(log_2(n))$, wenn n die Gesamtanzahl der Schlüsselwerte ist. Eine Verbesserung kann hierbei die k-näre Suche, mit Hilfe von SIMD-Operationen (Def. 2.8) bieten. Dabei werden $k-1$ Schlüssel gleichzeitig mit einem einzelnen Wert verglichen (s. Bsp. 2.5). Der Suchraum wird damit nicht in zwei Partitionen, sondern in k Partitionen zerlegt. Somit sinkt die Komplexität auf $O(log_k(n))$, was bei der aktuellen SIMD-Breite von 128 Bit und einem Datentyp von 8 Bit ein k von 17 ergibt und damit eine theoretische Reduzierung um den Faktor vier ($\frac{log_2(n)}{log_k(n)} = log_2(k) \approx 4,01$) bedeutet. Bereits 2002 haben Zhou und Ross in [174] die Verwendung von SIMD-Operationen bei der Implementierung von Datenbankoperatoren untersucht. Im Unterschied zu dieser Arbeit und der Arbeit von Schlegel et al. [148] werden darin bzgl. der Indexsuche nur linearsortierte Knoten mit 32 Bit Fließkommawerten untersucht.

3.4.1 Der k-näre Suchbaum

Ausgehend von dieser Beobachtung haben Schlegel et al. in [148] untersucht, wie man SIMD-Rechnerarchitekturen und die k-näre Suche nutzen kann, um die Suche in einem Index zu beschleunigen. Bei der aktuellen SIMD-Breite von 128 Bit können somit 16 8 Bit Werte, acht 16 Bit Werte, vier 32 Bit Werte oder zwei 64 Bit Werte gleichzeitig verglichen werden. Daraus folgt eine 17-näre Suche für 8 Bit Werte (bzw. respektive 9, 5 und ternäre Suche für 16, 32 sowie 64 Bit Werte). Die effiziente Verwendung von SIMD-Operationen setzt allerdings die Bedingung voraus, dass alle 16 (8,4,2) Werte gleichzeitig mit einem Befehl geladen werden können (vgl. Ergebnisse [148]). Was wiederum zur Konsequenz hat, dass die entsprechenden Werte direkt hintereinander im Speicher stehen müssen. Die Werteliste im Speicher bildet somit, nicht wie bei der binären Suche, eine lineare Ordnung ab, sondern linearisiert einen k-nären Suchbaum.

Definition 3.12 (k-näre Suchbaum) Ein k-närer Suchbaum besteht aus Knoten, in denen jeweils $k-1$ Werte gespeichert werden und innere Knoten auf k weitere Knoten verweisen. □

In [148] muss der Suchbaum für die Transformation aus der linearen Ordnung in die linearisierte k-näre Ordnung eine bestimmte Eigenschaft bzgl. der Menge an Elementen im Suchbaum aufweisen. Es wird davon ausgegangen, dass der Baum entweder ein perfekter k-närer Baum ist, also genau $k^h - 1$ Elemente aufweist mit $h \in \mathbb{N}$ oder ein kompletter Baum ist (eine leicht abgeschwächte Definition eines perfekten Baums aus [148]).

In dieser Arbeit wurde die Arbeit von Schlegel et al. aufgenommen und um die folgenden offenen Punkte erweitert:

(i) Verwendung der Idee zur k-nären Suche in vorhandenen Indexstrukturen, wie B^+-Bäumen.

(ii) Verbesserung des Verhaltens bei Änderungsoperationen.

(iii) Vergleich bzgl. der Möglichkeiten zur Linearisierung: Breiten- gegenüber Tiefensuche.

(iv) Verallgemeinerung auf eine beliebige Anzahl von Schlüsselwerten im Suchbaum.

(v) Evaluation aller vier Integer Datentypen (8,16,32 und 64 Bit), sowie Messung des Einflusses der Methode zur Bestimmung des SIMD-Vergleichsergebnisses (Bsp. 2.5).

3.4.1.1 Die k-näre Suche in B^+-Bäumen

Nach [23, 40] ist ein B-Baum wie folgt definiert:

Definition 3.13 (B-Baum) Seien $h \geq 0$ und $n \geq 1$ Natürliche Zahlen ($n, h \in \mathbb{N}$). Ein gerichteter Baum T ist aus der Klasse $\tau(n, h)$ von B-Bäumen, wenn T entweder leer ist ($h = 0$) oder die folgenden Eigenschaften besitzt:

(i) Jeder Pfad von der Wurzel zu jedem Blatt hat die gleiche Länge h, auch als Höhe des Baumes bezeichnet.

(ii) Jeder Knoten, außer der Wurzel und den Blättern, hat mindestens $n + 1$ Kinderknoten. Die Wurzel ist entweder ein Blatt oder hat mindestens zwei Kinderknoten.

(iii) Jeder Knoten hat maximal $2n + 1$ Kinderknoten. □

Der B^+-*Baum* speichert – im Gegensatz zum B-Baum – Daten (TIDs) nur in den Blättern [103]. Daraus folgt für die Suche nach dem Schlüsselwert v im B^+-*Baum*, dass für innere Knoten mit den sortierten Werten $(a_0, ..., a_i)$ mit $k \leq i \leq 2k$ die Position p bestimmt werden muss, so dass $a_{p-1} \leq v < a_p$ gilt. Dabei wird bei der Wurzel gestartet und der Pfad jeweils um den p-ten Kindknoten erweitert bis ein Blatt erreicht ist. Im Blatt muss dann explizit auf die Existenz von v geprüft werden. Im Nachfolgenden wird sich diese Arbeit o.B.d.A. allein mit der Suche in den inneren Knoten beschäftigen.

Um die k-näre Suche für einen B^+-*Baum* zu nutzen, wird jeder Knoten als unabhängiger k-närer Suchbaum angesehen werden (s. Abb. 3.9). Diese Eigenschaft ist speziell

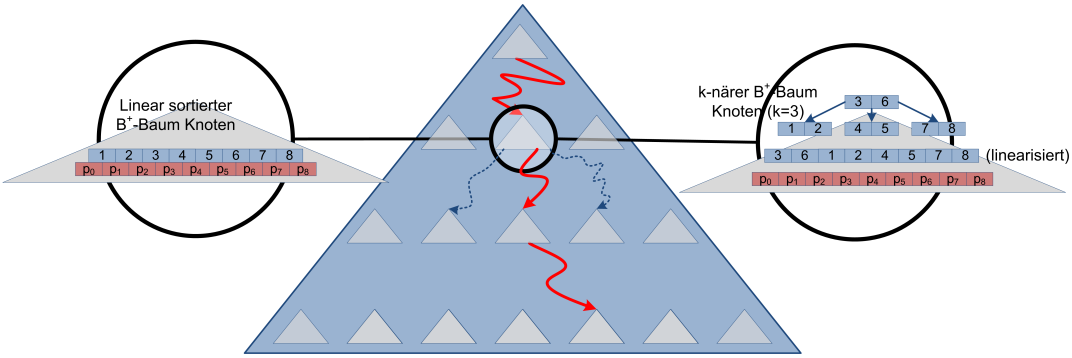


Abbildung 3.9: B^+-*Baum:* Knoten mit linearer Ordnung vs. k-närer Linearisierung

für Änderungsoperationen auf dem Index wichtig. Da damit die Veränderungen lokal begrenzt werden und nicht der komplette Baum neu aufgebaut werden muss. Bei der Transformation werden aus diesem Grund auch nur die Schlüsselwerte selbst nach einem k-nären Suchbaum linearisiert und Verweise auf andere Knoten bleiben unverändert und müssen daher bei Änderungsoperationen nicht neu sortiert werden (vgl. Abb. 3.9 rechts). Da aber die Anzahl der im Knoten vorhandenen Einträge jeden Wert zwischen n und $2n$ annehmen kann, ist die in [148] gestellte Bedingung nicht immer gegeben. Es bedarf somit einer Möglichkeit auch nicht perfekte oder komplette Suchbäume der k-nären Suche zugänglich zu machen. Zunächst sollen aber die beiden Möglichkeiten der Linearisierung vorgestellt werden.

3.4.1.2 Möglichkeiten der Linearisierung

Ein k-närer Suchbaum kann effizient aus der sortierten Liste der Schlüsselwerte gewonnen werden (Abb. 3.9 links). Dazu wird eine Transformationsformel benötigt, die jeder Position der sortierten Liste (p_L) einen Knoten und die Position im Knoten des k-nären Suchbaums zuordnet. Die Transformationsformel kann hierbei mit Hilfe der Breitensuche (BF) $P_{BF}(p_L)$ oder Tiefensuche (DF) $P_{DF}(p_L)$ umgesetzt werden. Im Folgenden werden beide Formeln für einen Datentyp D_m, mit der Länge von m Bits und einer SIMD-Operationsbreite von $|SIMD|$ Bits (Def. 2.8) angegeben. Dabei ist p_L die Position des Wertes in der linear sortierten Liste, k ist die Ordnung des Suchbaums mit $k = \frac{|SIMD|}{m} + 1$ und N lässt sich wie folgt berechnen: Für eine Liste von n Elementen werden mindestens $r = \lceil log_k n \rceil$ Ebenen eines k-nären Suchbaums benötigt. Seien r und k gegeben, dann berechnet sich N mit $N = k^r$ und gibt die maximale Anzahl von Elementen an $(N-1)$, die im Baum auf r Ebenen gespeichert werden können.

BF – Breadth-First Search

DF – Depth-First Search

Die Transformationsformel nach Breitensuche $P_{BF}(p_L)$, berechnet damit aus der Position $p_L = (0, ..., n-1)$ die Position im linerarisierten k-nären Suchbaum $(0, ..., N-1)$ und lässt sich wie folgt rekursiv definieren:

$$P_{BF}(p_L) = P_{BF}(p_L, 0)$$

$$P_{BF}(p_L, R) = \begin{cases} \frac{p_L+1}{S(R-1)}(k-1) & \\ \quad + \frac{(p_L+1) mod (S(R)k)}{S(R)} - 1 & \text{if } (p_L+1) mod S(R) = 0, \\ P_{BF}(p_L, R+1) & \\ \quad + k^R(k-1) & \text{sonst.} \end{cases}$$

Für die Transformationsformel nach Tiefensuche $P_{DF}(p_L)$ gilt:

$$P_{DF}(p_L) = P_{DF}(p_L, 0)$$

$$P_{DF}(p_L, R) = \begin{cases} \frac{(p_L+1) mod S(R-1)}{S(R)} - 1 & \text{if } (p_L+1) mod S(R) = 0, \\ P_{DF}(p_L, R+1) & \\ \quad +(k-1) & \\ \quad + \frac{(p_L+1) mod S(R-1)}{S(R)}(S(R)-1) & \text{sonst.} \end{cases}$$

mit $S(R) = \left\lfloor \frac{N}{k^{R+1}} \right\rfloor$.

Beispiel 3.15: Transformation eines Knotens mit Hilfe von P_{DF}

In diesem Beispiel soll am Beispiel des in Abbildung 3.9 dargestellten Knotens die Transformation in die linearisierte Form mit Hilfe der Tiefensuche vorgestellt werden. Dazu soll der Baumknoten acht 64 Bit Schlüsselwerte enthalten, woraus sich für k sich bei $|SIMD| = 128$Bit ein Wert von drei ergibt. Bei $n = 8$ beträgt die Höhe des ternären Baums $r = \lceil log_3 8 \rceil = 2$ und damit ergibt sich $N = 3^2 = 9$. Es handelt sich somit um einen perfekten Baum der Höhe zwei. Bei der Transformation wird für jedes $p_L = 0, ..., 7$ $P_{DF}(p_L)$ berechnet (s. Tab. 3.1 bzw. Abb. 3.9 rechts).

Bei der Suche nach dem Schlüsselwert $v = 6$ wird der transformierte B^+-*Baum-Knoten*, wie folgt durchsucht und $p = 6$ als Position für den Zeiger auf den nächsten Knoten im Pfad gefunden werden. Wie in Kapitel 2 in Beispiel 2.5 gezeigt wurde, werden zuerst die beiden Werte drei und sechs gleichzeitig in ein SIMD Register geladen und mit v vergleichen. Das Ergebnis ist in diesem Fall drei, d.h., der dritte k-nären Knoten der nächsten Ebene (7,8) muss weiter durchsucht werden. Das Ergebnis dieses Vergleiches ist Null. Mit dem Wissen, dass es der dritte Knoten auf der zweiten Ebene war, ergibt sich das korrekte Gesamtergebnis von sechs für p.

p_L	0	1	2	3	4	5	6	7
$P_{DF}(p_L)$	2	3	0	4	5	1	6	7

Tabelle 3.1: Umrechnung der Positionen

3.4.1.3 Transformation von Suchbäumen beliebiger Größe

Für die Nutzung des Konzepts der k-nären Suche in B^+-Bäumen ist es notwendig, keinerlei Anforderungen an die Anzahl der Schlüsselwerte im Suchbaum zu haben. Da allerdings bei der Suche mit Hilfe von SIMD pro Vergleich immer $k-1$ viele Elemente vorhanden sein müssen, wird ähnlich zu [148] vorgegangen.

Zunächst wird der größte, vorhandene Schlüsselwert S_{max} bestimmt, d.h., das letzte Element in der nach der linearen Ordnung sortierten Liste. Im Anschluss werden mit Hilfe der Transformationsformel die Schlüsselwerte in die linearisierte Ordnung eines k-nären Suchbaums gebracht. Danach werden alle nicht komplett gefüllten k-nären Knoten mit dem Schlüsselwert $S_{max}+1$ auf $k-1$ Elemente aufgefüllt (s. Abb. 3.10).

Bei der Suche nach v im Suchbaum muss dann zunächst überprüft werden, ob $v > S_{max}$ ist. Sollte dieses der Fall sein, muss die Suche übersprungen werden und wenn es sich um einen inneren Knoten handelt, muss dem $n+1$ Verweis des Knotens im B^+ Baum gefolgt werden. Für den Fall, dass es ein Blatt ist, ist v folglich nicht im Index vorhanden.

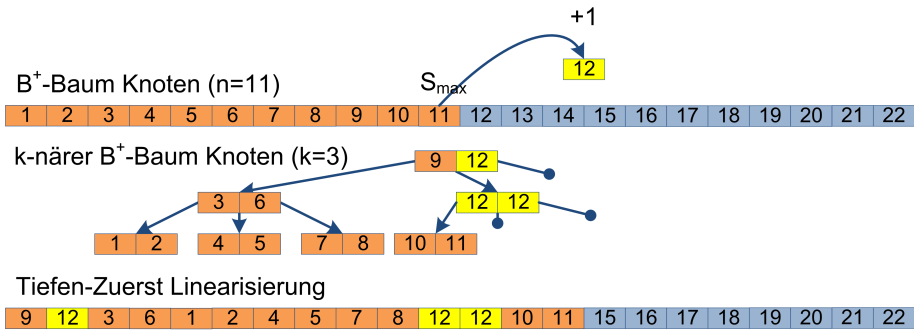


Abbildung 3.10: Linearisierung eines nicht kompletten k-nären Baums

3.4.1.4 Evaluierung der Verwendung der k-nären Suche im B^+-Baum

Zum Schluss soll die Evaluation aller vier Integer-Datentypen (8, 16, 32 und 64 Bit) sowie die Messung des Einflusses der Methode zur Bestimmung des SIMD-Vergleichsergebnisses vorgestellt werden. Bei dem Vergleich mit Hilfe von SIMD-Operationen werden zwei SIMD-Register miteinander verglichen. Das Ergebnis steht ebenfalls in einem SIMD-Register und stellt eine spezielle Bitmaske dar (vgl. Bsp. 2.5). Um nach dem Vergleich herauszufinden, an welcher Stelle im Baum weiter gesucht werden muss, wird das Ergebnis-SIMD-Register in ein 32 Bit Integer umgewandelt. Die gesetzten bzw. nicht gesetzten Bits dieser Zahl, geben dann das Ergebnis des Vergleichs wieder. Es muss also

das Bitmuster analysiert werden, um das eigentliche Vergleichsergebnis zu erhalten. Dieses kann z.B. über eine der drei Möglichkeiten aus Beispiel 2.5 geschehen (Bitschieben, Case-Anweisung, Spezialbefehl *popcnt*).

In Abbildung 3.11 ist ein Vergleich der Laufzeiten bzgl. dieser drei Möglichkeiten dargestellt. Gemessen wurde ein k-närer Knoten für einen 8 Bit Datentyp. Die drei Kategorien Single, 5MB und 100MB geben die vorhandene Datenmenge an und sollen den Einfluss der Speicherherarchie (s. Abschn. 2.1.1.1) untersuchen.

Es ist zu sehen, dass die Methode über den speziellen Prozessorbefehl *popcnt* die besten Ergebnisse liefert, dazu kommt eine sehr starke Unabhängigkeit von der Datenmenge, d.h. der Örtlichkeit der Daten. Der Unterschied entsteht hauptsächlich dadurch, dass mit dem *popcnt* Befehl 16 bedingte Sprünge eliminiert werden und es somit zu keinerlei Pipelineflushes (s. Abschn. 2.1.2.2) kommt. Die Messungen wurden auch für die Datentypen mit 16, 32 und 64 Bit gemacht. Dabei nimmt der Leistungsvorteil mit größeren Datentypen stark ab, da z.B. bei einem 64 Bit Typ nur zwei bedingte Sprünge notwendig sind.

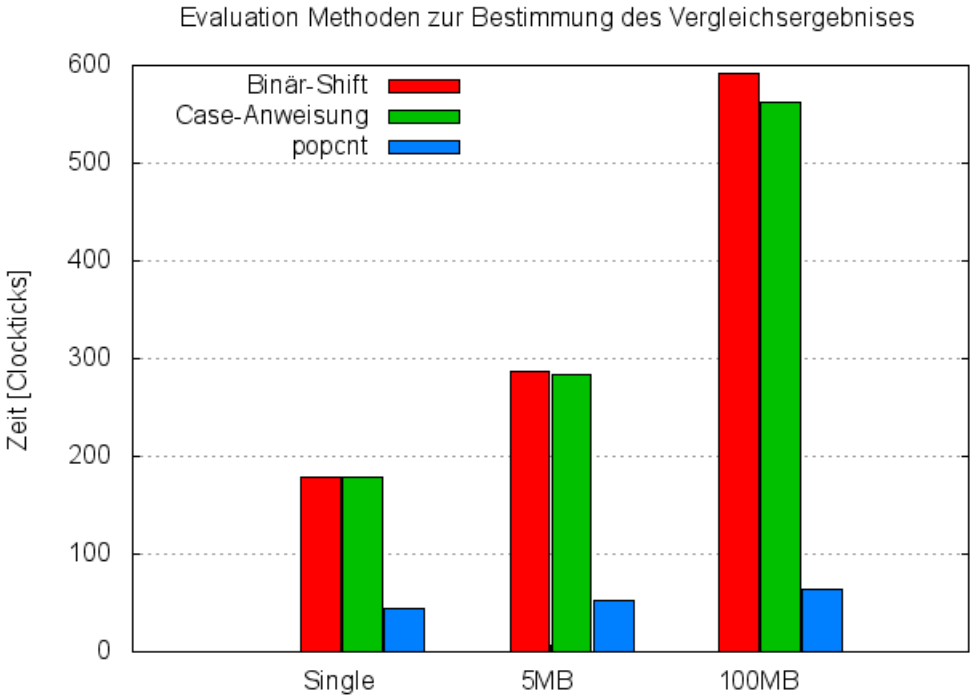


Abbildung 3.11: Vergleich der Methoden zur Bestimmung des Vergleichergebnisses

Zuletzt sollen die Ergebnisse bzgl. der unterschiedlichen Datentypen, verschiedener Elementanzahl und der beiden Linearisierungsmöglichkeiten vorgestellt werden. Tabelle 3.2 stellt dazu die Konfiguration der B^+-*Baum-Knoten* dar, dabei sind: k die Ordnung des k-nären Suchbaums, N_L die maximale Anzahl von Elementen im linear sortierten

B^+-*Baum-Knoten*, N_S die maximale Anzahl von Werte im k-nären Suchbaum eines Knotens, sowie r die Tiefe des k-nären Suchbaums (s. Abschn. 3.4.1.2).

Datentyp	**k**	N_L	N_S	**r**	**N**	**sizeof(Node)**	**CacheLines**
int8_t	17	254	256	2	289	2296	2
int16_t	9	404	408	3	729	4056	7
int32_t	5	338	344	4	625	4096	11
int64_t	3	242	242	5	243	3880	16

Tabelle 3.2: Eigenschaften der B^+-Baumknoten

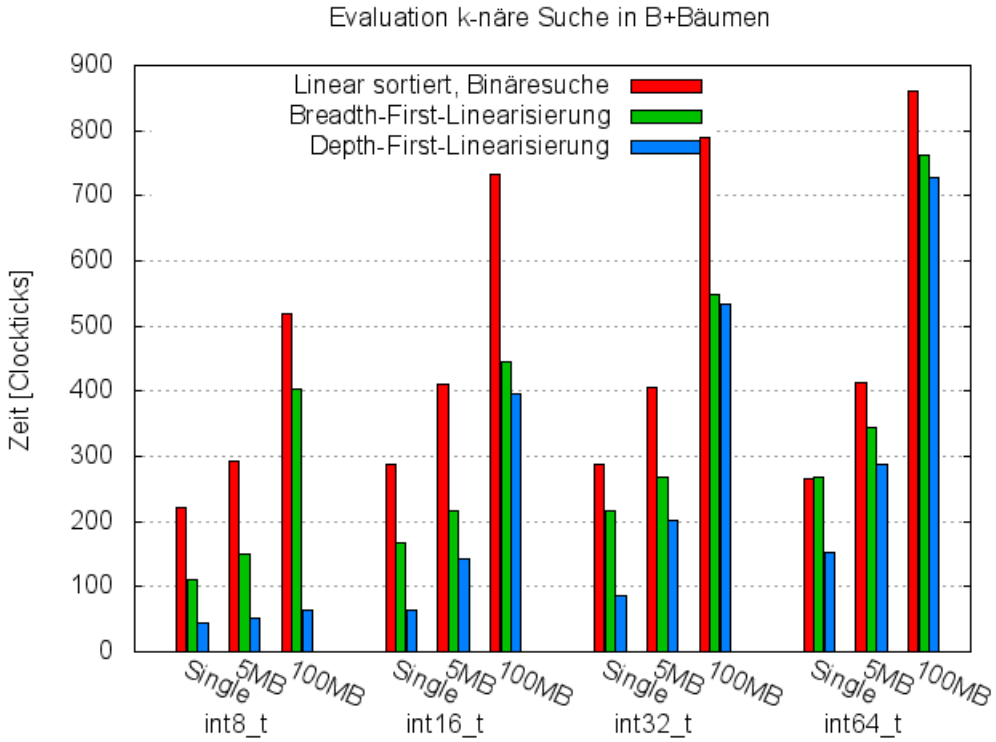


Abbildung 3.12: Vergleich: k-näre Suchbäume

Abbildung 3.12 stellt die Laufzeiten für die Suche nach einem Schlüssel v im Knoten dar. Als Basis dient die Binärsuche in einem normalen B^+-*Baum-Knoten* (rot). Dazu kommen die Messungen für die k-näre Suche mit Hilfe von SIMD und der PopCount-Methode auf einem mit Tiefensuche (blau) und Breitensuche (grün) linearisierten Knoten. Die einzelnen Datentypen *intX_t* haben dabei eine Größe von X Bit. Die Kategorien *Single*, *5MB* und *100MB* repräsentieren verschiedene Gesamtgrößen des Baumes. Mit den verschiedenen Kategorien sollte das Verhalten bzw. der Speicherhierarchie analysiert werden. Bei den Datentypen (8 Bit und 16 Bit), bei denen der Wertebereich keine

entsprechende Menge an unterschiedlichen Werten aufweist, wurde eine entsprechende Menge simuliert.

Die Messungen zeigen, dass die Linerarisierung über Tiefensuche grundsätzlich besser ist als die beiden anderen Linerarisierungsarten. Ebenfalls ist erwartungsgemäß festzustellen, dass der Leistungsgewinn durch die Parallelisierung der Suche für den 8 Bit Datentyp (int8_t) am größten ist und – wie bereits zuvor vorgestellt – relativ unabhängig von der Datenmenge ist. Dieses Ergebnis lässt sich wie folgt begründen: (i) Durch die kleine Breite von 8 Bit könnten mit einem mal 16 Werte gleichzeitig verglichen werden. Und (ii) erlaubt die kleine Breite auch eine effektivere Verwendung des Caches, es kommt somit zu einer besseren Cachenutzung. Das Problem mit einem Datentyp von 8 Bit ist, dass maximal 256 verschiedene Werte darstellbar sind und somit sein Anwendungsbereich eher gering ist. Um trotzdem die Vorteile nutzen zu können und auch mehr als 256 Werte darstellen zu können, wurde in dieser Arbeit eine neue Art von Index entwickelt.

3.4.2 Der segmentierte Index

In diesem Abschnitt soll mit Hilfe einer neuen Indexstruktur, **dem segmentierten Index**, eine Möglichkeit vorgestellt werden, die Leistungsvorteile des k-nären Suchbaums auf dem 8 Bit-Datentyp auch für größere Datentypen nutzen zu können. Der auf L Bit segmentierte Index ($SegIdx_L$) auf dem Datentyp D_m, mit der Länge von m Bit, speichert Schlüsselwerte aus D_m und zugehörige Tupelmengen und ist wie folgt definiert:

Definition 3.14 (Segmentierte Index) $SegIdx_L$ ist ein Baum mit $r = \frac{m}{L}$ Ebenen $(E_0, ..., E_{r-1})$. Die Ebene E_0 umfasst genau einen Knoten und bildet somit die Wurzel des Baums, dessen Knoten auf jeder Ebene Teilschlüssel in der Länge von L Bit enthalten. Jeder Knoten kann k $(1 \leq k \leq 2^L)$ Teilschlüssel enthalten, dabei ist bei Knoten der Ebenen E_i $(0 \leq i \leq r-2)$ die Anzahl der Verweise auf einen Knoten der nächsten Ebene E_{i+1} genauso groß wie die Anzahl von Teilschlüsseln im Knoten. Die Knoten der Ebene E_{r-1} enthalten genauso viele Verweise auf Tupelmengen wie Teilschlüssel. Der i-te Verweis gehört dabei immer zum i-ten Teilschlüssel und umgekehrt. □

Operationen auf dem segmentierten Index

Der Baum speichert einen Schlüssel $S[b_{m-1}...b_0]$, indem S in r viele Segmente $S_0, ..., S_{r-1}$ zerlegt wird, wobei $S_i[b_{L-1}...b_0] = S[b_{(i+1)L-1}...b_{iL}]$ $(0 \leq i \leq r-1)$ ist. Das i-te Segment S_i dient dann als Schlüssel für die Ebene E_i. Bei der Suche wird der Baum von der Wurzel E_0 hin zur Ebene E_{r-1} durchlaufen. Existiert auf einer Ebene der entsprechende Teilschlüssel nicht, ist der Schlüssel S nicht im Index enthalten. Wird dagegen der Baum bis zur letzten Ebene durchlaufen, gibt der gefundene Verweis die Tupelmenge für S wieder. Die Operationen des Einfügens und Löschens werden ähnlich zur Suche definiert, werden in dieser Arbeit aber nicht weiter ausgeführt.

Beispiel 3.16: Instanz eines segmentierten Indexes

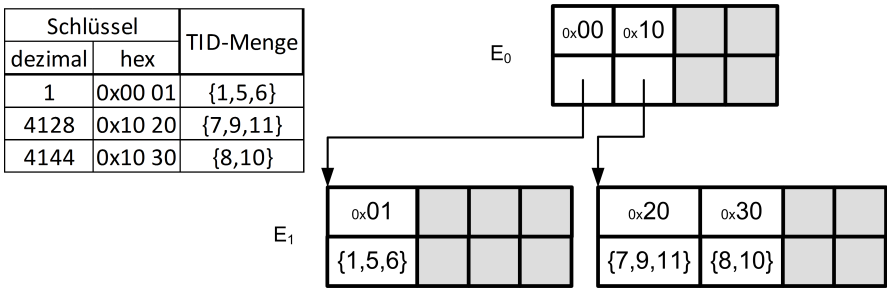


Abbildung 3.13: Instanz eines segmentierten Indexes

Auf Abbildung 3.13 ist eine Beispielinstanziierung für einen auf 8 Bit segmentierten Index gegeben (rechts). Die Tabelle (links) zeigt Schlüssel, deren hexadezimalen Wert, sowie die zugehörige TID-Menge, die im Index gespeichert wird. Der Datentyp der Schlüssel soll hier $m = 16$ Bit groß sein, damit ergeben sich $r = \frac{16}{8} = 2$ Ebenen. Die Teilschlüssel im Index sind durch ihren hexadezimal Wert dargestellt.

Es soll die TID-Menge für den Schlüssel 4129 ($0x1021$) gesucht werden, dazu wird im Wurzelknoten (Ebene E_0) mit dem Teilschlüssel $S_0 = 0x10$ gesucht und der entsprechende Verweis auf einen Knoten der nächsten Ebene E_1 gefunden. In diesem Knoten wird daraufhin mit dem Teilschlüssel $S_1 = 0x21$ gesucht, der aber keinen passenden Wert findet. Somit existiert für den Schlüssel 4129 kein Indexeintrag.

Im Weiteren wird o.B.d.A. von einer Segmentierung auf 8 Bit ($L = 8$) ausgegangen und einem Datentyp von 64 Bit ($m = 64$). Der segmentierte Index hat damit 8 Ebenen, jeder Knoten enthält maximal 256 Werte, der gesamte Index kann allerdings 2^{64} Schlüsselwerte speichern. Die Teilschlüssel der Knoten dieses segmentierten Indexes können nun mit Hilfe eines 17-nären Suchbaums linearisiert und entsprechend durchsucht werden. Bei einer SIMD-Breite von 128 Bit reichen somit maximal zwei Vergleichsoperationen pro Knoten, also maximal 16 Vergleichsoperationen für den gesamten Index. Bei einem ternären Index sind es dagegen $\lceil log_3 n \rceil$ viele Operationen; bei der binären Suche $\lceil log_2 n \rceil$. Damit führt der segmentierte Index ab einer Anzahl von $n \geq 3^{16}$ (ternäre Suchbaum) bzw. $n \geq 2^{16}$ (binäre Suche) gespeicherten Schlüsselwerten weniger Vergleichsoperationen aus. Dazu kommt der größere Leistungsgewinn durch die Verwendung von 8 Bit SIMD-Vergleichsoperationen. Der segmentierte Index hat aber noch drei weitere Vorteile:

(i) Da die gemeinsamen Präfixe der Schlüssel nicht immer wiederholt werden müssen, stellt er eine Art von Komprimierung dar. Man kann den segmentierten Index als eine Art von Präfixbaum auf Bitebene interpretieren, er stellt somit eine Erweiterung zu bekannten Präfixindizes dar [24, 132, 145, 16]. Goetz Graefe beschreibt in [60] ein Verfahren, um gemeinsame Präfixe innerhalb von B^+-Bäumen zu verwenden, um Platz zu sparen und spricht dabei von „Prefixtruncation“.

(ii) Durch die feste Anzahl von Ebenen wird der Anzahl von Suchoperationen, Seiten-

und Speicherzugriffen eine feste obere Grenze gesetzt.

(iii) Da die Ebenen immer fest definierte Ausschnitte der Schlüssel verwenden, führen Einfüge- und Löschoperationen im k-nären Suchbaum nur zu einer knotenlokalen Restrukturierung, der restliche Baum ist nicht betroffen.

Neben seinen Vorteilen weist der segmentierte Index einen großen Nachteil bei der gleichmäßigen Verteilung der gespeicherten Schlüssel über die gesamte Domäne auf. Dann kann es im schlechtesten Fall dazu kommen, dass alle Knoten der oberen Ebenen voll gefüllt sind, aber die Knoten der letzten Ebene jeweils nur einen Wert enthalten. Man würde damit unnötig viel Speicherplatz verbrauchen. In diesen Fällen kann das Problem dadurch beseitigt werden, dass die Zuordnung der Segmente und der Ebenen vertauscht wird. Eine ideale Verwendungsmöglichkeit des segmentierten Indexes stellen Indizes auf fortlaufenden Nummerierungen dar, wie bei Surrogaten (Def. 3.5) oder TIDs (Def. 3.6). In diesen Fällen füllt sich der Index langsam von links nach rechts.

Eine mögliche Verbesserung des segmentierten Index ist die ausschließliche Speicherung von solchen Ebenen, die mindestens zwei unterschiedliche Werte enthalten. Eine alternative Verbesserung ist das direkte Lesen des entsprechenden Verweises sobald alle 256 Werte des 17-nären Suchbaums vorhanden sind. In diesem Fall kann auch der entsprechende Suchbaum gelöscht werden und der Knoten des segmentierten Indexes geht in eine Art Hash über.

Evaluierung des segmentierten Indexes

Für die Evaluation des segmentierten Indexes, wurde eine prototypische Implementierung angefertigt, die mit einem B^+-*Baum* auf einem 64 Bit Datentyp verglichen wurde. Abbildung 3.14 zeigt den Faktor der Verbesserung gegenüber der B^+-*Baum*-Variante mit linear sortierten Schlüssel und Binärsuche in Abhängigkeit von der Tiefe des Baumes. Zu sehen sind die beiden k-nären Varianten des B^+-*Baums* in Tiefen- und Breitensuche, sowie zwei Implementierungen des segmentierten Indexes. Der optimierte segmentierte Index bezieht sich dabei auf die zuvor beschriebene Optimierung bzgl. des Entfernens von Ebenen.

Es ist zuerkennen, dass mit dem Wachstum der Tiefe des B^+-*Baums* die Verbesserung des segmentierten Indexes zu nimmt, während die k-nären Varianten eine von der Tiefe des Baumes unabhängige Verbesserung generieren. Ebenfalls ist die Verbesserung durch die Optimierung zu erkennen. Für die Schlüsselsuche wurde eine Verbesserung um einen Faktor von etwa 14 gemessen; dazu kam eine Reduktion um den Faktor acht beim Platzbedarf.

Zusammenfassung

Im Kapitel 3 wurde zuerst das Relationenmodell als Modell der konzeptuellen Schicht zusammen mit einer erweiterten relationalen Algebra vorgestellt. Darauf aufbauend wurden verschiedene Möglichkeiten für das physische Datenmodell dargeboten, speziell die

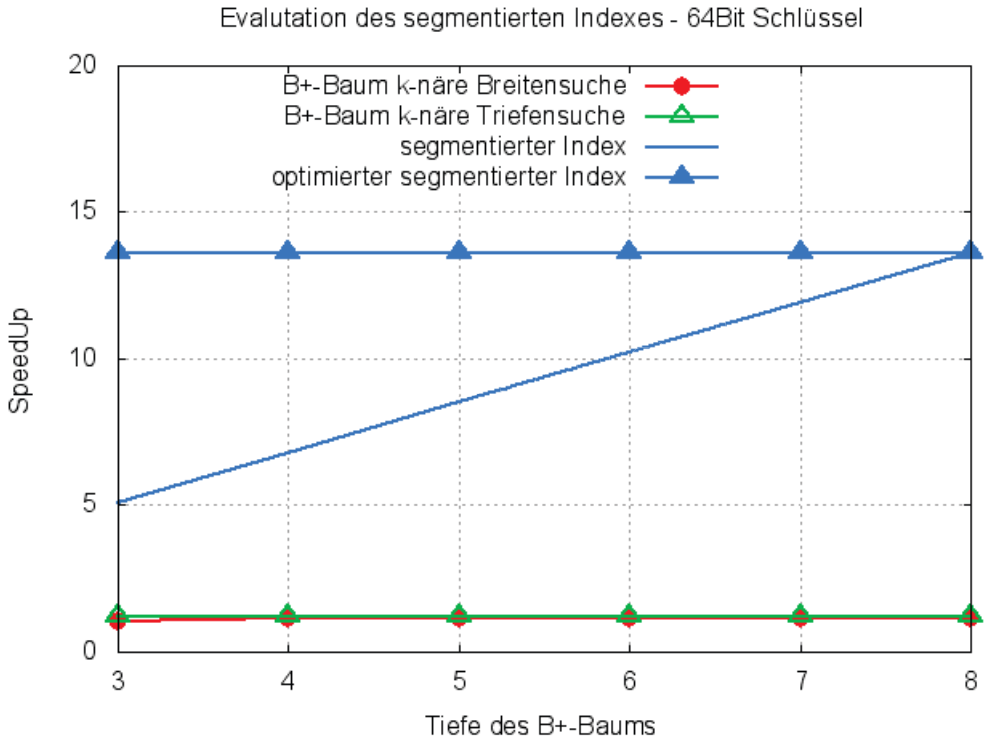


Abbildung 3.14: Evaluation des segmentierten Indexes

vertikale Partitionierung. Im dritten Abschnitt wurde dieses Modell als Grundlage des physisches Datenmodell des Frameworks ((s. Abschn. 1.3)) definiert, dazu kamen globale Tupelidentifikatoren (gTIDs) und ihre Anwendungsbereiche für verschiedene Indizes (Join-, Domänen-, Bereichsindizes). Zur Repräsentation von Mengen von gTIDs wurde das Konzept von Bit-Strings vorgestellt. Weiter wurden verschiedene Operatoren und die Verwendung der abstrakten Operatoren aus dem Hardwaremodell (s. Abschn. 2.3) exemplarisch vorgestellt. Den Abschluss des Kapitels bildete der Abschnitt zur Verwendung von SIMD-Operationen bei der Suche in Indizes. Dabei wurde als erstes der k-näre Suchbaum vorgestellt und im Anschluss mit dem segmentierten Index eine neue Indexstruktur präsentiert. Zuerst wurde dazu der segmentierte Index definiert, danach Vor- und Nachteile diskutiert und letztendlich Ergebnisse der Evaluierung vorgestellt. Dabei konnte der segmentierte Index mit einem Leistungsgewinn um einen Faktor von etwa 14 ein sehr gutes Leistungsverhalten aufzeigen.

4 Anfrageausführung

„Beauty is the ultimate defense against complexity“

David Gelernter (1955–)

Das Kapitel 4 beschäftigt sich mit der Anfrageausführung in RDBMSen. Die zentrale Frage ist, wie mit Hilfe eines gegebenen QEPs die Antwortmenge an Tupeln aus der Datenbank erzeugt werden kann. Im Vordergrund stehen dabei speziell die verschiedenen Modelle zur Anfrageausführung. Aus diesem Grunde werden im ersten Teil Grundlagen der Anfrageausführung vorgestellt, darunter verschieden Strategien der Ausführung, sowie Möglichkeiten zur parallelen Ausführung eines QEPs. Im zweiten Teil werden verschiedene Modelle für die serielle als auch für die parallele Ausführung präsentiert und verglichen. Im dritten und letzten Teil des vierten Kapitels wird ein im Rahmen dieser Arbeit entwickeltes Anfrageausführungsmodell vorgestellt, welches den dritten Teil des Frameworks darstellt. Ziel ist es ein Modell für die Anfrageausführung zu entwickeln, das für Mehrkern-Rechnerarchitekturen geeignet ist und einen großen Grad an Parallelität bietet. Dazu wird zuerst die Ausführung von Anfragen ohne Joins betrachtet, wie z.B. die erste Anfrage des TPC-H-Benchmarks; anschließend folgt die Erweiterung um die Joinverarbeitung.

Anfrage Q_1 des TPC-H-Benchmarks

Bei der ersten Anfrage des TPC-H-Benchmarks (s. Abschn. 1.1.2) werden verschiedene Aggregate bzgl. der Gruppen aus *returnflag* und *linestatus* gebildet. Als Filterbedingung werden nur die Tupel in das Ergebnis aufgenommen, die in *shipdate* einen Datumswert zwischen dem 1.12.1998 und *[DELTA]* Tagen davor haben. Die Anfrage ist gut als Beispiel geeignet, da auf Grund ihrer Join-freiheit relativ einfach ist. Im Benchmark wird der Wert für *[DELTA]* so gewählt, dass sich zwischen 95% und 97% aller Tupel qualifizieren [162]. Im Beispiel 4.1 ist eine vereinfachte Variante dieser Anfrage dargestellt, in ihr wird zugunsten der besseren Lesbarkeit[1] nur eine Summe und die Gesamtanzahl von Tupeln pro Gruppe gebildet.

[1] auch Präfixe wurden weggelassen

Beispiel 4.1: Vereinfachte TPC-H-Anfrage Q_1

```
select
    returnflag,
    linestatus,
    sum(quantity) as sum_qty,
    count(*) as count_order
from
    lineitem
where
    shipdate <= date '1998-12-01'
        - interval '[DELTA]' day
group by
    returnflag,
    linestatus;
```

4.1 Grundlagen der Anfrageausführung

Die Anfrageausführung ist der letzte Teil der Anfragebearbeitung (Def. 1.4), dabei wird der durch die Anfrageoptimierung erzeugt QEP (Def. 1.5) über der Datenbank ausgeführt. Für die Ausführung existieren zwei unterschiedliche Strategien, in Bezug auf die Richtung, in der der QEP abgearbeitet wird.

Definition 4.1 (Auftragsgetriebene Ausführung) Die **auftragsgetriebene Ausführung** wird von oben nach unten ausgeführt (engl. Top-Down). Dabei beginnt der oberste Operator (die Wurzel) die Ausführung des gesamten QEPs und fordert damit Eingangsdatenströme von seinen Kindern an. Diese Anforderung wird dann durch die Operatoren bis zu den Blättern des QEPs hinunter gereicht. □

Definition 4.2 (Datengetriebene Ausführung) Die **datengetriebene Ausführung** kennzeichnet sich damit, dass die Ausführung von unten nach oben verläuft (engl. Bottom-Up). Die bereitstehenden Daten bestimmen den Kontrollfluss, so dass Operatoren gestartet werden, sobald Daten oder Tupel in den Eingangsströmen bereit liegen. Der Kontrollfluss geht immer von den Blättern des QEPs zur Wurzel, er wird hierbei nicht durch ineinander geschachtelte Funktionsaufrufe realisiert, sondern über gemeinsame Puffer oder auf Basis von Nachrichten, die zwischen den Operatoren ausgetauscht werden. Zudem existiert eine zentrale Instanz (engl. Scheduler), die die einzelnen Operatoren startet und beendet. □

Parallele Anfrageausführung

In parallelen RDBMSen wird die Intraquery-Parallelität (Def. 1.7) genutzt, um die Antwortzeit zu verringern. Dazu wird die Auswertung des QEPs über mehrere Prozessoren oder Rechner verteilt. Der QEP wird dazu in disjunkte Teile – sogenannte **Fragmente** (s. Def. 4.5) – zerlegt, die dann durch die verschiedenen Prozessoren parallel verarbeitet werden können. Intraquery-Parallelität kann in zwei Arten unterteilt werden:

Definition 4.3 (Intraoperator-Parallelität) Bei der **Intraoperator-Parallelität** wird ein einzelner Operator parallelisiert, wie z.B. beim parallelen Sortieren oder Suchen [111]. □

Ein Operator, der auf einer großen Tabelle arbeitet, kann parallelisiert werden, indem die gleiche Operation auf verschiedenen Teilen der Tabelle parallel ausgeführt wird. Dieser Art von Intraoperator-Parallelität, wird **partitionierte Parallelität** genannt. Abbildung 4.1 zeigt einen QEP, bei dem die Intraoperator-Parallelität für die drei Join-Operatoren verwendet wird. Das Hauptaugenmerk bei der Intraoperator-Parallelität liegt auf: (i) Wie kann eine Operation auf verschiedenen Datensätzen ausgeführt werden? Und (ii), wie müssen die Daten partitioniert werden, um auf ihnen arbeiten zu können?

Es ist somit für die Intraoperator-Parallelität notwendig, die sequentiellen Operatoren in kleine Bestandteile aufzubrechen, die dann auf den Partitionen unabhängig voneinander arbeiten können. Im Abschnitt 3.3.3 wurden für die Selektion (Bsp. 3.11) und die Aggregation (Bsp. 3.12) zwei parallelisierbare Operatoren auf dem Datenmodell vorgestellt.

Definition 4.4 (Interoperator-Parallelität) Werden mehrere verschiedene Operatoren eines QEPs parallel ausgeführt, wird darauf mit **Interoperator-Parallelität** referenziert. Es werden zwei Formen der Interoperator-Parallelität unterschieden: (i) **Vertikale** oder auch **Pipeline-Parallelität**. Sowie die (ii) **horizontale Parallelität**, die auch als **unabhängige Parallelität** bezeichnet wird [111]. □

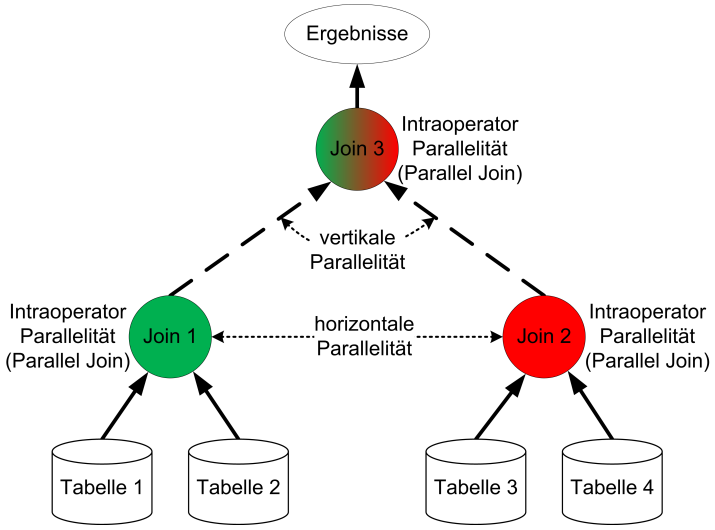


Abbildung 4.1: Formen der Intraquery-Parallelität

Bei der **horizontalen Parallelität** werden Operatoren, die nicht voneinander abhängen, parallel ausgeführt, so z.B. die beiden Join-Operatoren $Join1$ und $Join2$ auf Abbildung 4.1. Der Grad der möglichen Parallelität ist dabei stark durch die Gesamtanzahl der Operatoren und ihrer Abhängigkeiten begrenzt.

In der **vertikalen Parallelität** laufen zwei voneinander abhängige Operatoren parallel. Dabei konsumiert Operator B, die Ausgabedaten von Operator A, ohne das A bereits das komplette Ergebnis berechnet hat. In Anfrageplänen wird die vertikale Parallelität, um sie von der sequentiellen Ausführung zu unterscheiden, oft durch eine gestrichelte Kante zwischen zwei Operatoren dargestellt. In Abbildung 4.1 besteht z.B. zwischen dem Operator $Join3$ und $Join1$, sowie zwischen den Operatoren $Join3$ und $Join2$, eine vertikale Parallelität. Benötigt ein Operator das vollständige Ergebnis eines seiner Kindoperatoren, wird diese Kante als **blockierende Kante** bezeichnet.

Definition 4.5 (Fragment) Ein **QEP-Fragment** ist ein zusammenhängender Teilgraph des QEPs, bei dem alle Operatoren mit Hilfe von vertikaler Interoperator-Parallelität (Def. 4.4) parallelisiert werden können. Fragmente enthalten keine *blockierenden Kanten*; stattdessen verbinden *blockierende Kanten* einzelne Fragmente eines QEPs miteinander. □

4.2 Modelle der Anfrageausführung

Im Folgenden sollen Modelle zur Realisierung der Anfrageausführung vorgestellt werden. Dazu wird zunächst das Iteratormodell als typischer Vertreter der seriellen, auftragsgetriebenen Ausführung vorgestellt. Dazu kommen Erweiterungen des Iteratormodells für die parallele Ausführung. Anschließend wird die Anfrageausführung in MonetDB erläutert, darunter auch eine besondere Art, die vektorbasierte Anfrageausführung in MonetDB/X100, gefolgt von der datengetriebenen Ausführung im Gamma-System. Zuletzt werden die vorgestellten Modelle miteinander verglichen und auf ihre Eignung für Anfrageausführung auf Mehrkern-Rechnerarchitekturen geprüft.

4.2.1 Das Iteratormodell

ONC – Open-Next-Close

Das Iteratormodell (ONC) ist eine oft verwendete Realisierung des Prinzips der auftragsgetriebenen Ausführung. Dabei besitzen alle Operatoren die Funktionen *Open*, *Next* und *Close*. Die Ausführung eines QEP beginnt mit einem *Open* auf der Wurzel. Ein über *Open* aufgerufener Operator ruft nach der Initialisierung – Öffnen von Dateien, Allokation von Speicher – wiederum alle seine Kinder bzgl. des Operatorbaums über *Open* auf.

Nach der erfolgreichen Initialisierung aller Operatoren des QEPs beginnt die eigentliche Ausführung. Dazu wird die Wurzel mit *Next* aufgerufen. Der *Next*-Aufruf liefert immer das nächste Ergebnis des entsprechenden Operators zurück. Ein Operator kann wiederum seine Kinder mit *Next* aufrufen, um von ihnen neue Ergebnisse zu erhalten. Wurden alle Ergebnisse über die Wurzel abgerufen, folgt der abschließende *Close*-Aufruf, der ähnlich zum *Open* die Operatoren schließt und verwendete Ressourcen wieder freigibt.

Das ONC-Prinzip bildet eine Produzenten-Konsumenten Verarbeitung nach, bei dem der aufrufende Operator der Konsument und der aufgerufene Operator der Produzent ist. Für eine detaillierte Darstellung wird auf [59] bzw. [67] verwiesen. Das ONC-Prinzip macht die Komposition von Anfrageplänen einfach, da sich die verschiedenen Operatoren

alle über eine gemeinsame Schnittstelle ansprechen lassen. Zumeist stellt das Ergebnis eines *Next*-Aufrufs ein Tupel dar, dabei wird dann von **Tupel-at-a-time** gesprochen.

Ohne weitere Maßnahmen ist das ONC-Prinzip inhärent seriell, da immer nur ein Operator arbeitet. Für die parallele Anfrageausführung wurde ein spezieller Operator eingeführt, der *Exchange*-Operator. Er dient zum Informationsaustausch zwischen zwei QEP-Fragmenten (Def. 4.5) [58].

Eine Erweiterung des Iteratorkonzepts für Mehrkern-Rechnerarchitekturen beschreiben Acker et al. in [4], wobei sie einen *ASYNC*-Operator beschreiben, der ähnlich zum *Exchange* Operator ist, nur das er inhärent Intraoperator-Parallelität darstellen kann und ordnungserhaltend ist.

4.2.2 Das Anfrageausführungsmodell von MonetDB

Das Ausführungsmodell von MonetDB (s. Abschn. 3.2.2) folgt nicht dem *Tupel-at-a-time*-sondern dem **Column-at-a-time-Konzept**, demzufolge berechnet und materialisiert der produzierende Operator immer das komplette Ergebnis, bevor er es an den Konsumenten weiter gibt [27]; es gibt somit nur blockierende Kanten im QEP. Der Vorteil liegt unter anderem in der besseren Möglichkeit ILP-Techniken (s. Abschn. 2.1.2.2) auszunutzen und damit ein besseres Leistungsverhalten zu erreichen. Auf der anderen Seite erzeugt die Materialisierung der kompletten Zwischenergebnisse einen starken Zuwachs an Speichertransfers zwischen Hauptspeicher und dem Prozessor und kann damit auch zu Leistungsverlusten führen.

Bei der Ausführung erstellen die einzelnen Operatoren BATs (s. Abschn. 3.2.2), in denen sie die Ergebnisse ablegen. Temporäre BATs können eine besondere Form haben und werden dann als **Pivottabellen** bezeichnet. Die Form ist dabei so, dass der Head vom Typ *VOId* ist und der Tail vom Type *OId*. Der Tail stellt immer Referenzen auf Tupel dar, die im Ergebnis des jeweiligen Operator liegen. Die Ergebnistupel werden erst im allerletzten Schritt zusammengesetzt und damit in die konzeptuelle Sicht, dem Relationenmodell, transformiert. Die eigentliche Ausführung basiert auf der Verarbeitung von BATs, dazu sind in MonetDB verschiedenste Operatoren definiert. Für eine umfassende Darstellung der Operatoren und ihrem Zusammenspiel wird auf [28] verwiesen. Im Folgenden wird kurz die Ausführung der vereinfachten TPC-H-Anfrage Q_1 (Bsp. 4.1) durch MonetDB vorgestellt.

Beispiel 4.2: Anfrage Q1 in MonetDB

```
1  _1:bat[:oid,:date] :=
2     sql.bind("sys","lineitem","shipdate",0);
3  _8 := mtime.date_sub_sec_interval(1998-12-01,7776000:lng);
4  _9 := algebra.thetauselect(_1,_8,"<=");
5  _12 := algebra.markT(_9,0@0);
6  _13 := bat.reverse(_12);
7  _14:bat[:oid,:str] :=
8     sql.bind("sys","lineitem","returnflag",0);
9  _16 := algebra.leftjoin(_13,_14,nil:lng);
10 (ext23,grp21) := group.new(_16);
```

```
11 _20:bat[:oid,:str] :=
12    sql.bind("sys","lineitem","linestatus",0);
13 _22 := algebra.leftjoin(_13,_20,nil:lng);
14 (ext29,grp27) := group.derive(ext23,grp21,_22);
15 _25 := bat.mirror(ext29);
16 _26 := algebra.join(_25,_16);
17 _27 := algebra.join(_25,_22);
18 _28:bat[:oid,:lng] :=
19    sql.bind("sys","lineitem","quantity",0);
20 _30 := algebra.leftjoin(_13,_28,nil:lng);
21 _28:bat[:oid,:lng] := nil:BAT;
22 _31 := algebra.selectNotNil(_30);
23 _32:bat[:oid,:lng] := aggr.sum(_31,grp27,_25);
24 _33:bat[:oid,:wrd] := aggr.count(grp27,grp27,_25);
25 _34 := sql.resultSet(4,1,_26);
26 sql.rsColumn(_34,"sys.lineitem",
27    "l_returnflag","char",1,0,_26);
28 sql.rsColumn(_34,"sys.lineitem",
29    "l_linestatus","char",1,0,_27);
30 sql.rsColumn(_34,"sys.lineitem",
31    "sum_qty","decimal",15,2,_32);
32 sql.rsColumn(_34,"sys.lineitem",
33    "count_order","wrd",64,0,_33);
34 _50 := io.stdout();
35 sql.exportResult(_50,_34);
```

MAL – MonetDB Assembler Language

Der überstehende Quelltext stellt die interne Darstellung bzw. die Abarbeitung der vereinfachten Anfrage Q_1 (Bsp. 4.1) mit Hilfe einer internen Sprache (MAL) von MonetDB dar und folgt dem nachstehenden Ablauf: In den Zeilen 1 bis 6 wird zuerst die Spalte *shipdate* aus der Relation *LineItem* ausgewählt (1-2) und mit Hilfe der Selektionsbedingung gefiltert (3-4). Die Zeilen 5 und 6 dienen dazu die eigentliche Pivottabelle (*_13*) zu erstellen, sie ist vom Typ (*VOId*,*OId*) und enthält in der Tail-Spalte *OIds* von allen selektierten Tupel aus *LineItem*.

Als nächstes wird die Gruppierung (Zeilen 7-10) auf *returnflag* berechnet und anschließend die Gruppierung auf *linestatus* (Zeilen 11-15). Das Ergebnis sind zwei BATs. Zum einen wieder eine Pivottabelle (*grp27*), die jedem selektierten Tupel eine Gruppe zuordnet; zum anderen eine neue BAT (*_25*), die die einzelnen Gruppen darstellt und jeweils auf einen Repräsentanten – den entsprechenden Attributwert – verweist.

In den Zeilen 18 bis 24 werden die beiden Aggregationen bzgl. der einzelnen Gruppen berechnet. Das Ergebnis sind wiederum zwei BATs, die jeder Gruppe den entsprechenden Wert der Aggregation zuordnen. Zum Schluss (Zeile 25-35) werden die eigentlichen Ergebnistupel erzeugt, indem die einzelnen BATs über die Gruppenidentifikatoren gejoint werden. Abbildung 4.2 stellt den Ausführungsgraphen und die Abhängigkeiten zwischen den einzelnen Operatoren und ihren Ergebnissen dar.

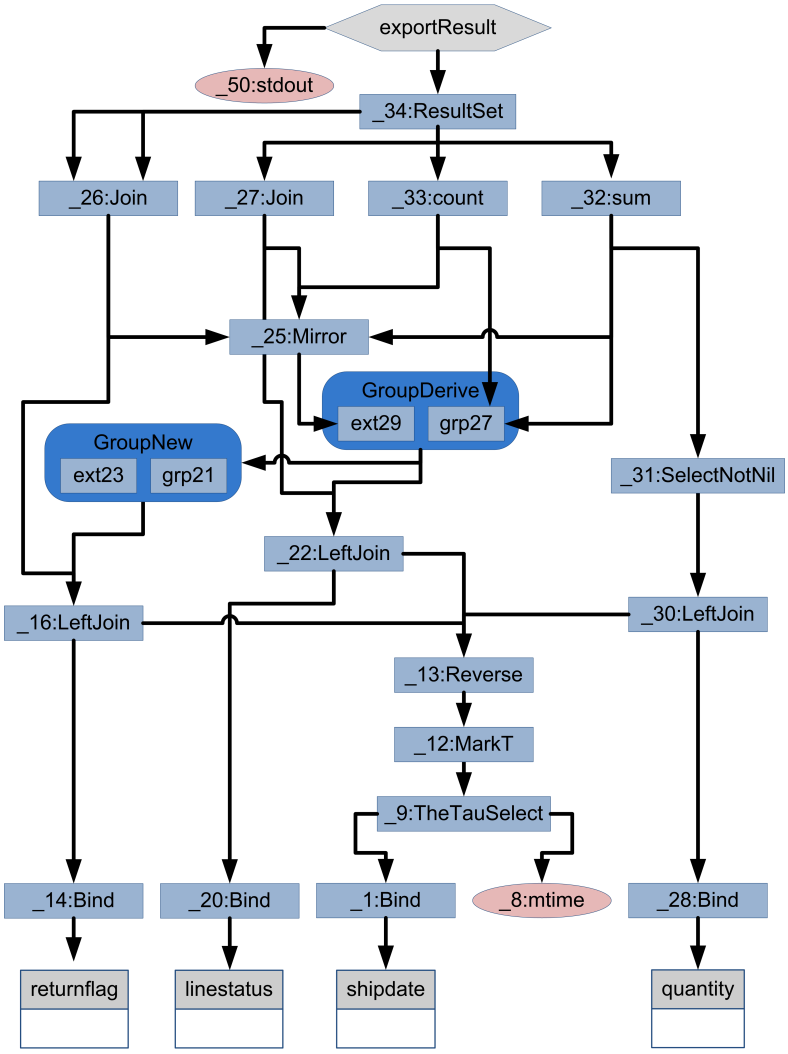


Abbildung 4.2: Ausführungsgraph von MonetDB für die vereinfachte TPC-H-Anfrage Q_1

Parallele Anfragebearbeitung in MonetDB

Die Column-at-a-time-Verarbeitung erlaubt keine vertikale Interoperator-Parallelität, allerdings ist eine Intraoperator-Parallelität, sowie eine horizontale Interoperator-Parallelität möglich. Die horizontale Interoperator-Parallelität, also die parallele Ausführung mehrerer unabhängiger Operatoren, ist in MonetDB implementiert, dabei führen verschiedene Threads einzelne Operatoren parallel aus.

Die Intraoperator-Parallelität wurde in der Masterarbeit von Marcin Zukowski [175] untersucht, darin betrachtet er, wie sich die einzelnen Operatoren zerlegen lassen. Die Parallelisierung wird über die Partitionierung der Eingangs-BAT in N Teile ermöglicht, die dann auf N verschiedenen Prozessoren parallel ausgeführt werden. Im Anschluss werden die N Ergebnis-BATs wieder zusammengefügt. Die Anzahl von Partitionen N ist dabei eine feste Systemgröße, die keinerlei Rücksicht auf die Auslastung oder andere Merkmale nimmt. Zukowski beschreibt Möglichkeiten zur Parallelisierung für die Selektion, die Aggregation, das Sortieren, sowie für den Join. Er geht speziell auf den Radix-Cluster-Join [115] ein, eine besondere Art eines partitionierten Hash-Joins.

Beispiel 4.3: Horizontale Interoperator-Parallelität

Für die Anfrage im Beispiel 4.1, deren serielle Ausführung zuvor beschrieben wurde (Bsp. 4.2), sind auf Abbildung 4.3 (vgl. Abb. 4.2) die einzelnen Operatoren mit Intraoperator-Parallelität grün gekennzeichnet; gelb eingefärbte Knoten können nicht parallelisiert werden. Mit Hilfe der horizontalen Interoperator-Parallelität können alle Operatoren auf einer Ebene, also zwischen zwei braunen horizontalen Linien, parallel verarbeitet werden. Die rot markierten Kanten stellen mögliche vertikale Interoperator-Parallelitäten dar, die allerdings mit dem Column-at-a-time-Modell nicht ausgenutzt werden können. Im Abschnitt 4.3.1 wird diese Arbeit darauf zurückkommen und eine mögliche Lösung zeigen.

4.2.3 MonetDB/X100 Vektorverarbeitung

Wie bereits zuvor beschrieben, hat die Column-at-a-time-Verarbeitung in MonetDB den Nachteil, der durch zusätzliche Speichertransfers erzeugt wird. MonetDB/X100 löst dieses Problem, indem es die Vorteile der beiden Welten Column-at-a-time und Tupel-at-a-time vereint. Dazu wurde ein vektorisiertes Anfrageausführungsmodell entwickelt, das auf dem Iteratormodell beruht. Anstatt allerdings immer nur einzelne Tupel hin und her zureichen, wird ein Vektor also eine Menge von Tupeln über jeden *Next*-Aufruf weitergegeben. Die Funktionen sind dabei so geschrieben, dass sie vom Compiler automatisch vektorisiert (Def. 2.30) werden können. Die Größe der Vektoren ist so ausgelegt, dass die Ergebnisse immer in den Cache passen. Mehr Details und Beispiele sind in [27] zu finden.

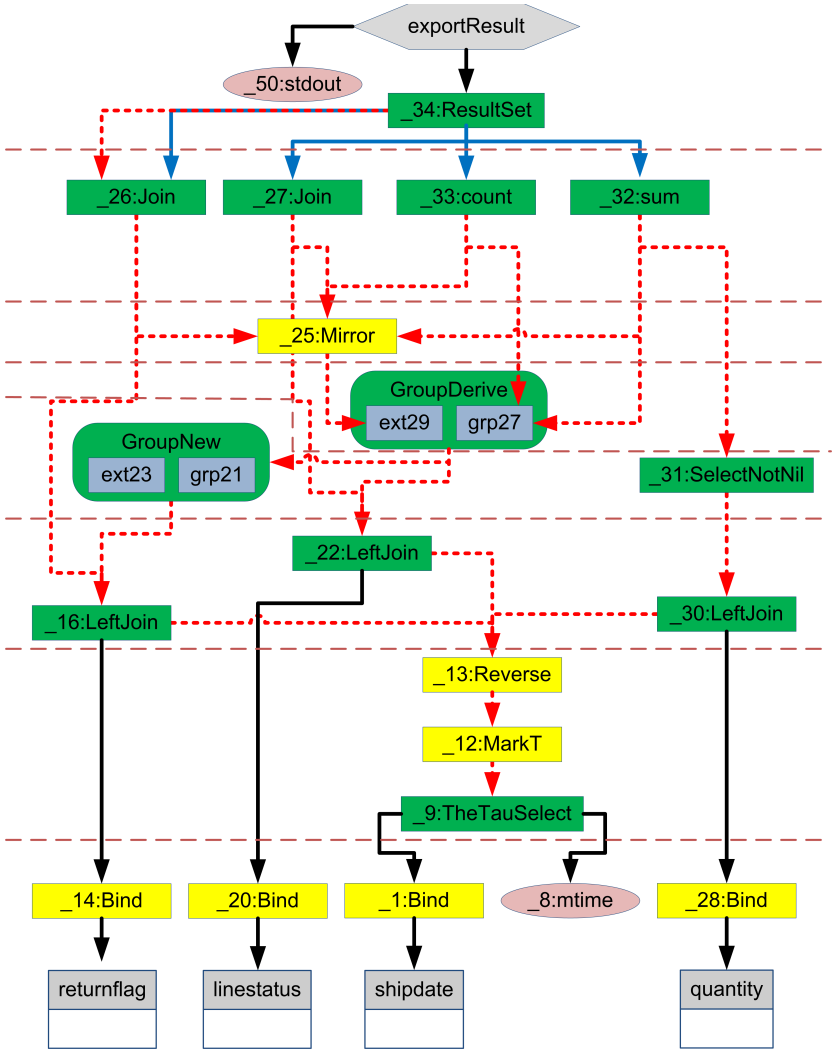


Abbildung 4.3: Paralleler Ausführungsgraph von MonetDB für die vereinfachte TPC-H-Anfrage Q_1

4.2.4 Anfrageausführung in Gamma

Die Anfrageausführung in Gamma ist ein Vertreter der datengetriebenen Ausführung. Das Gamma-System [45] stellt ein paralleles RDBMS auf einer Shared-Nothing-Architektur (s. Abschn. 1.1.3) dar. Dabei wird für jede Anfrage ein eigener Scheduler gestartet, der dann die Kontrolle über die Abarbeitung des QEPs übernimmt, indem er Operatoren bzw. deren Prozesse startet bzw. beendet. Der Kontrollfluss wird über Nachrichten dargestellt, die der Scheduler an die Operatoren sendet bzw. die Operatoren an den Scheduler senden. Innerhalb des QEPs wird die Pipeline-Parallelität ausgenutzt, indem sich die Operatoren Datenpakete zusenden. Intraoperator-Parallelität wird mit Hilfe von

Relationspartitionierung oder dynamischer Repartionierung erreicht. Für eine tiefergehende Betrachtung wird auf [46] verwiesen.

4.2.5 Vergleich der Ausführungsmodelle

Im Folgenden sollen kurz die vier vorgestellten Anfrageausführungsmodelle verglichen werden. Tabelle 4.1 fasst die Eigenschaften zusammen und bewertete ihre Parallelisierbarkeit.

	Iterator	MonetDB	X100	Gamma
Modell	Auftrags-getrieben			Daten-getrieben
Verarbeitungseinheit	Tupel	Spalten	Tupelmengen	
Möglichkeiten zur Parallelisierung				
– Intraoperator	o	++	+	++
– horz. Interoperator	+	++	+	++
– vert. Interoperator	o	x	o	++

x– keine, o– schlecht, +– gut, ++– sehr gut

Tabelle 4.1: Vergleich der Ausführungsmodelle

Die Vor- und Nachteile der vier Modelle sind in Tabelle 4.2 dargestellt. Beim Vergleich der verschiedenen Ausführungsmodelle ist zu erkennen, dass das Iteratormodell und das vektorbasierte Iteratormodell sehr gut für Einzelprozessoren geeignet sind. Dagegen scheinen die Column-at-a-time-Verarbeitung von MonetDB bzw. die datengetriebene Verarbeitung von Gamma durch ihre Möglichkeiten der Parallelisierung, besser für Mehrkern-Rechnerarchitekturen geeignet zu sein. Im nächsten Abschnitt wird daher zunächst das MonetDB-Modell auf eine taskbasierte Parallelisierung erweitert, gefolgt von der Präsentation eines Ausführungsmodell basierend auf der datengetriebenen Verarbeitung.

Modell	**Vorteile**	**Nachteile**
Iterator	• Geringe Notwendigkeit für die Zwischenspeicherung von Ergebnissen. • Leicht zu implementieren.	• Schwer bzw. aufwendig zu parallelisieren. • Kaum Möglichkeiten für die Verwendung von SIMD. • Schlecht adaptierbar (s. Abschn. 5.2).
MonetDB	• Gut parallelisierbar, da die einzelnen Zwischenergebnisse sowieso materialisiert werden müssen. • Möglichkeit der Verwendung von SIMD Befehlen.	• Materialisierung aller Zwischenergebnisse. • Keine Möglichkeit für die vertikale Parallelität.

Modell	Vorteile	Nachteile
X100	• Gegenüber des Tupel-at-a-time-Ansatzes auch gut Intraoperator parallelisierbar. • Möglichkeit der Verwendung von SIMD Befehlen. • Pipelining, dadurch eine Verringerung der Notwendigkeit für die Zwischenspeicherung von Ergebnissen, im Gegensatz zu Column-at-a-time.	• Schwer bzw. aufwendig zu parallelisieren. • Kaum Möglichkeiten für die Verwendung von SIMD. • Schlecht adaptierbar (s. Abschn. 5.2).
Gamma	• Sehr gut parallelisierbar. • Möglichkeit der Adaption von QEPs zur Laufzeit.	• Viel Synchronisationsaufwand für gemeinsame Puffer. • Zusätzlicher Aufwand für die Organisation der Abarbeitung.

Tabelle 4.2: Vor- und Nachteile der verschiedenen Modelle

4.3 Die Schicht der Anfrageausführung

In diesem Abschnitt wird der Teil der Anfrageausführung des Frameworks (s. Abschn. 1.3) vorgestellt (s. Abb. 4.4). Dazu wird im ersten Teilabschnitt gezeigt, wie das Ausführungsmodell von MonetDB unter Verwendung der Arbeit von Marcin Zukowski auf die taskbasierte Ausführung (Def. 2.20) erweitertet werden kann. Damit wird eine dynamische Parallelisierung auf Intraoperator- als auch auf horizontaler Interoperator-Ebene erreicht. Im zweiten Teilabschnitt wird die Anfrageausführung mit Hilfe von Bit-Strings und die parallele Ausführung von Operationen auf Bit-Strings vorgestellt. Im letzten Teilabschnitt wird ein datengetriebenes Ausführungsmodell vorgestellt, das im Framework verwendet wird. Dabei wird auf Ergebnisse und Techniken aus den ersten beiden Teilabschnitten zurückgegriffen. Ziel ist es, dass alle Operatoren gleichzeitig laufen, sowohl auf horizontaler als auch auf vertikaler Ebene; dabei wird über den Grad der Intraoperator-Parallelität eine Lastbalanzierung erreicht.

4.3.1 Erweiterung von MonetDB um das Taskkonzept

Dieser Abschnitt stellt die Parallelisierung mit Hilfe des Taskkonzepts für die Anfrageausführung in MonetDB vor. Zuerst wird die Intraoperator-Parallelität auf Basis des Taskkonzepts präsentiert, im Anschluss wird gezeigt, wie sich mit Hilfe von Taskgraphen (Def. 2.29) sehr einfach horizontale Parallelitäten darstellen lassen und wie mit Hilfe des Hardwaremodells – speziell unter der Verwendung von Belegungen (Def. 2.38) – eine optimale Ausnutzung der vorhandenen Ressourcen ermöglichen lässt.

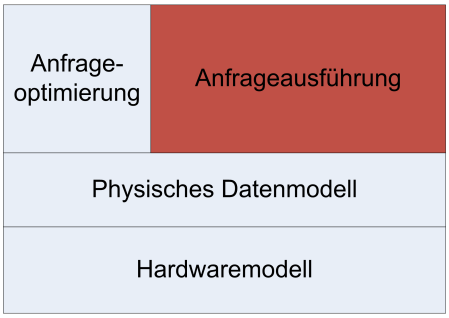


Abbildung 4.4: Framework: Anfrageausführung

4.3.1.1 Taskbasierte Intraoperator-Parallelität

In diesem Abschnitt wird eine Erweiterung des Ansatzes von Marcin Zukowski um das Taskkonzept (Def. 2.20) und damit die dynamische Zerlegung vorgestellt. Die Vorteile bei der Verwendung des Taskkonzepts sind: (i) seine implizite Synchronisation, (ii) eine optimale Ausnutzung der vorhandenen Ressourcen und (iii) die implizite Lastverteilung durch Verwendung des Taskgraphen und Taskstealing (s. Abschn. 2.2.4.3). Die Erweiterungen sollen exemplarisch am Beispiel des *Select* Operators von MonetDB vorgestellt werden.

Beispiel 4.4: Select Operator in MonetDB

```
1  BAT * selectEQOperator_Int(BAT * in, int val)
2  {
3      BAT * rc = BATnew(TYPE_oid, TYPE_int, estSize);
4      BUN p = 0, l;
5      BATiter bi = bat_iterator(in);
6      for(l=BUNlast(in), p=BUNfirst(in); p<l; ++p)
7      {
8          int * v = BUNtloc(bi, p);
9          if(*v == val)
10         {
11             bunInsert(rc, v);
12         }
13     }
14     return rc;
15 }
```

Der Quelltext zeigt die Implementierung eines *Select-Operators* für den Datentyp *Integer*. Darin wird zuerst eine neue BAT erzeugt (Zeile 3) und anschließend durch die Eingabe-BAT *in* iteriert und jeder Wert auf Gleichheit mit dem übergebenen Wert *val* geprüft (Zeile 8). Im Fall der Gleichheit wird der Wert und seine *OId* in die neue BAT *rc* eingefügt (Zeile 10).

Wie bereits vorgestellt, schlägt Zukowski in seiner Arbeit [175] vor, die Eingabe-BAT in N Teile zu zerlegen und auf jedem Teil die entsprechende Funktion auszuführen. Anschließend müssen die entstandenen Ergebnis-BATs wieder zusammengesetzt werden. Die Zerlegung basiert auf der *Slice-Operation* von MonetDB, die eine BAT virtuell zerlegt. Diese Operation ist sehr effizient, da nur einige wenige Pointer und Werte verändert werden; die Tupelwerte bleiben unberührt.

Für die **taskbasierte Intraoperator-Parallelität** wird ähnlich zum Ansatz von Zukowski vorgegangen. Allerdings wird eine BAT nicht in eine feste Anzahl von Stücken zerlegt, sondern rekursiv geteilt, bis eine bestimmte Granularität erreicht ist. Dann erst wird die so entstandene Task abgearbeitet. In der taskbasierten Version findet die Parallelisierung nicht auf MAL-Ebene statt, sondern auf Funktionsebene in der Implementierung selbst. Dazu wird das Konstrukt *parallel_for* aus den TBBs (s. Abschn. 2.2.4.3) verwendet, welches das rekursive Splitten, die Erstellung des Taskgraphen und die abschließende Reduktion übernimmt (s. Bsp. 4.5 Zeile 3). Die Selektion (Bsp. 4.4 Zeile 12-16) wird durch den abstrakten Operator *ABSTRACT_SEL* (vgl. Bsp. 3.11) aus dem Hardwaremodell des Frameworks implementiert. Mit Hilfe dieser Technik können viele auf einfachen Schleifen basierende Operatoren effizient parallelisiert werden.

Beispiel 4.5: Parallelisierung des Select Operators

```
1  BAT * selectEQOperator_Int_Parallel(BAT * in, int val,
2     int Grainsize){
3     BAT * rc = parallel_for(
4           blocked_range<BUN>(BUNfirst(in),BUNlast(in)),
5           NULL,
6           [](const blocked_range<BUN>& r,
7                 BAT * init)->BAT *{
8              BAT * rcP = BATnew(TYPE_oid,TYPE_int,
9                 r.end()-r.begin());
10             BATiter bi = bat_iterator(in);
11             BUN offset = r.begin(), end = r.end();
12             BUN skip = ABSTRACT_SEL.LEN();
13             while(offset < end){
14                ABSTRACT_SEL.OP(in,offset,val,rcP);
15                offset += skip;
16             }
17             return rcP;
18          },
19          []( BAT * left, BAT * right )-> BAT * {
20             appendBAT(left,right);
21             BATfree(right);
22             return left;
23          }, GrainSize );
24    return rc;
25 }
```

4.3.1.2 Taskbasierte horizontale Interoperator-Parallelität

Auch die horizontale Interoperator-Parallelität lässt sich mit Hilfe von Tasks und Taskgraphen deutlich einfacher darstellen, als wenn einzelne Operatoren explizit auf Threads verteilt werden. Dabei wird der QEP (s. Abb. 4.2) mit Hilfe von Tasks nachgebildet. Speziell im MonetDB Ausführungsmodell, bei dem ein Operator immer erst komplett ausgewertet wird, bieten Tasks und Taskgraphen eine sehr einfache aber elegante Möglichkeit, um die Abhängigkeiten zu modellieren und die Ausführung zu steuern.

Jeder Operator im QEP stellt eine einzelne **virtuelle Task** dar, die erst dann ausgeführt werden kann, wenn alle Tasks, von denen sie abhängt, also auf die eine Kante im Taskgraphen zeigt, abgearbeitet sind. Über den Taskgraphen wird somit der Kontrollfluss dargestellt, aber nicht der eigentliche Datenfluss. Der Datenfluss ist immer noch über gemeinsame Variablen (BATs) für die Ein- bzw. Ausgabe definiert. Die Verwendung des Taskkonzepts macht hierbei eine Analyse, welche Operatoren parallel ausgeführt werden können, überflüssig.

Dabei können die einzelnen virtuellen Tasks, bei ihrer Ausführung wiederum neue Tasks erstellten und damit die Intraoperator-Parallelität modellieren. Da aber alle durch die Intraoperator-Parallelität entstandenen Tasks als Elterntask die virtuelle Task des Operators haben, ist auch hier eine Analyse bzw. explizite Synchronisation unnötig.

4.3.1.3 Ressourcenverwaltung für die parallele Ausführung

Bisher wurden die Möglichkeiten der parallelen Ausführung von QEPs in MonetDB besprochen. Darunter wurde auch gezeigt, wie abstrakte Operatoren des Hardwaremodells genutzt werden können, um die Ausführung auf die Umgebung (z.B. den Prozessor) anzupassen. Im Kapitel 2 wurde das Konzept von Belegungen (Def. 2.38) auf dem Hardware-Graphen (Def. 2.32) eingeführt, um die Ausführung bzw. die Lokalität der Ausführung von Operatoren zu bewerten und zu steuern. Im Folgenden soll gezeigt werden, wie diese Belegungen und die Ausführung eines QEPs zusammenkommen.

Bei der Erzeugung des QEP, während der Anfrageoptimierung (s. Abschn. 5.3), wird jedem Operator im QEP eine Belegung zugeordnet, die festlegt welche und wie viele Ressourcen der Hardware der Operator verwenden darf. Bei der Ausführung bekommt die jeweilige virtuelle Task, die einen Operator darstellt, die entsprechende Belegung zugewiesen. Sobald die Task dann ausgeführt wird, werden die einzelnen Bedingungen umgesetzt. Wird ein Operator mit Hilfe von Intraoperator-Parallelität ausgeführt, können auch die einzelnen Tasks, die unterhalb der virtuellen Task angeordnet sind, auf die entsprechende Belegung zugreifen und sie umsetzen.

Der **Taskscheduler** ist für die Umsetzung der Lokalität der Ausführung zuständig und legt dabei fest welche Task eines Operator auf welchem Hardware-Thread (Def. 2.15) ausgeführt wird. Außerdem muss er dafür Sorge tragen, dass die einzelnen Workerthreads, die jeweils an einen einzelnen Hardware-Thread gebunden sind, nur Tasks ausführen bzw. „stehlen“, die auch ihrem Hardware-Thread zugeordnet sind.

Bei der Realisierung durch das TBB-Framework (s. Abschn. 2.2.4.3) war eine direkte Zuordnung von Tasks an einzelne Hardware-Threads oder Mengen von Hardware-

Threads nicht möglich. Dem Zufolge wurde bei der Evaluation nicht auf die Lokalität der Ausführung eingegangen. Eine Erweiterung der TBBs war im Rahmen dieser Arbeit nicht betrachtet worden, stattdessen wird die Umsetzung als Teil von zukünftigen Arbeiten gesehen (s. Abschn. 6.3).

Evaluation taskbasierte Parallelität in MonetDB

Im Zuge dieser Arbeit wurde die taskbasierte Intraoperator-Parallelität sowie die taskbasierte, horizontale Interoperator-Parallelität in MonetDB evaluiert. Dazu wurden die zuvor in Abschnitt 4.3.1 beschriebenen Konzepte (s. Abschn. 4.3.1) implementiert und an Hand der TPC-H-Anfrage Q_1 auf folgende Punkte hin untersucht: (i) die Skalierbarkeit bzgl. der Anzahl von Prozessoren und der damit erreichbare *SpeedUp* (Def. 1.8) und (ii) die Anzahl von zur Verfügung stehenden, lauffähigen Tasks. Anhand der Ergebnisse aus Punkt zwei sollten Aussagen getroffen werden, wie das Verhalten auf einem System mit einer größeren Anzahl von Hardware-Thread auszieht.

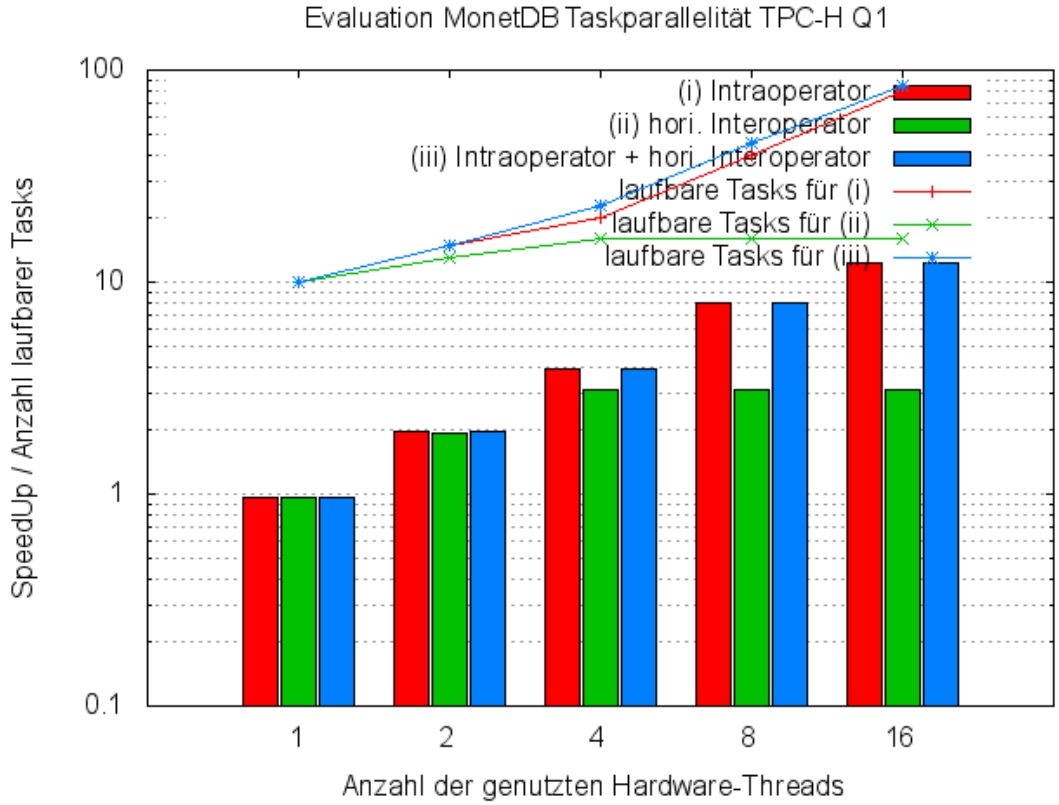


Abbildung 4.5: Auswertung der taskbasierten Parallelität bzgl. der TPC-H-Anfrage Q_1 in MonetDB

Abbildung 4.5 zeigt die Ergebnisse der Messungen bzgl. der verschiedenen Implementierungen und einer unterschiedlichen Anzahl von Hardware-Threads auf dem Testsys-

tem (s. Abschn. 2.1.5.4). Dargestellt sind zum einen der *SpeedUp* (Säulendiagramm) und zum anderen die relative Anzahl von zur Verfügung stehenden lauffähigen Tasks. Der *SpeedUp* wurde aus dem Quotienten der Zeit der Ausführung mit N Threads und der taskbasierten Implementierung und der Zeit für einen Thread und der ursprünglichen Implementierung berechnet. Die Threadanzahl von eins liefert somit den Zusatzaufwand, den die taskbasierte Implementierung mit sich bringt; er liegt zwischen zwei bis drei Prozent.

Bei der Implementierung wurden drei Fälle evaluiert: (i) reine Intraoperator-Parallelität (rot), (ii) reine horizontale Interoperator-Parallelität (grün) und (iii) die Kombination aus (i) und (ii) (blau). Auf der Abszisse sind die Anzahl an verwendeten Hardware-Threads aufgetragen (1,2,4,8,16). In diesem Zusammenhang sei darauf hingewiesen, dass es sich bei dem Testsystem um zwei Prozessoren mit je 4 Kernen handelt, die mit Hilfe von SMT je zwei Hardware-Threads parallel ausführen können. Es ist somit für den Schritt von 8[2] auf 16 Threads keine Verdoppelung des *SpeedUps* anzunehmen (s. Abschn. 2.1.5).

Die Messungen zeigen für die reine horizontale taskbasierte Parallelisierung eine relativ schlechte Skalierung mit steigender Anzahl von Hardware-Threads. Die Begründung liegt in der limitierten Anzahl von unabhängigen Operatoren und damit der begrenzten Anzahl von zur Verfügung stehenden Tasks. Für den Fall der reinen taskbasierten Intraoperator-Parallelität kann man einen nahezu idealen (linearen) *SpeedUp* sehen. Die Ausnahme ist der Fall von 16 Hardware-Threads, die Begründung hierfür wurde bereits zuvor gegeben.

Für die Kombination der taskbasierten Intraoperator- und horizontalen Interoperator-Parallelität konnte auf Grund der maximalen Auslastung der Prozessorleistung durch die Intraoperator-Parallelität kein weiterer *SpeedUp* verzeichnet werden. Wichtig ist aber, dass die Anzahl von zur Verfügung stehenden, lauffähigen Tasks noch einmal deutlich ansteigt. Diese Beobachtung lässt den Schluss zu, dass auch bei einer größeren Menge an zur Verfügung stehenden Hardware-Threads, ein guter *SpeedUp* erreichbar ist.

4.3.2 Anfrageausführung mit Hilfe von Bit-Strings

Als eines der zentralen Bestandteile des Frameworks wurden im Abschnitt des physischen Datenmodells Bit-Strings (Def. 3.11) eingeführt und Beispiele für Operatoren auf Bit-Strings vorgestellt (s. Abschn. 3.3.3). In diesem Teilabschnitt soll nun die Anfrageausführung mit Hilfe von Bit-Strings vorgestellt werden. Dazu wird zuerst am Beispiel der vereinfachten TPC-H-Anfrage Q_1 (Bsp. 4.1) eine mögliche Ausführung durch Verwendung von Bit-Strings vorgestellt. Der Fokus liegt dabei speziell auf dem Gruppierungsoperator und den Aggregationsoperatoren.

Bei komplexen Filterbedingungen können während der Anfrageausführung mit Hilfe von Bit-Strings große Teile des QEPs aus Bit-String-Operatoren bestehen. Daher ist für die Anfrageausführung im Framework eine effiziente Abarbeitung insbesondere von Operatoren, die als Eingabe und Ausgaben einen oder mehr Bit-Strings haben, unab-

[2] bei 1-8 Threads wurde kein SMT verwendet

dingbar. Im zweiten Teil werden daher verschiedene Methoden der parallelen Ausführung von **Bit-String-Operatorbäumen** vorgestellt werden.

Definition 4.6 (Bit-String-Operatorbaum) Ein **Bit-String-Operatorbaum** ist ein Operatorbaum, bei dem jeder Operator[3] einen oder mehrere Bit-Strings als Eingabe entgegennimmt und einen oder mehrere Bit-Strings als Ausgabe produziert. □

4.3.2.1 Ausführung der vereinfachten TPC-H-Anfrage Q_1

Die Ausführung der vereinfachten Anfrage Q_1 (Bsp. 4.1) des TPC-H-Benchmarks soll nachfolgend unter Verwendung von Bit-Strings anhand des QEPs von Abbildung 4.6 vorgestellt werden.

Die beiden *Group-Operatoren* durchlaufen die Spalten *returnflag* bzw. *linestatus* und erzeugen für jeden Attributwert einen Bit-String, der alle Tupel für den jeweiligen Attributwert repräsentiert. Der Selektionsoperator *Select* selektiert alle Tupel, die im Attribut *shipdate* einen der Anfrage entsprechenden Wert aufweisen und repräsentiert sie ebenfalls in einem Bit-String. Für den Fall, dass auf einem dieser Attribute ein passender Index definiert ist, können die Operatoren durch Indexzugriffe ersetzt werden (s. Abschn. 3.3.1).

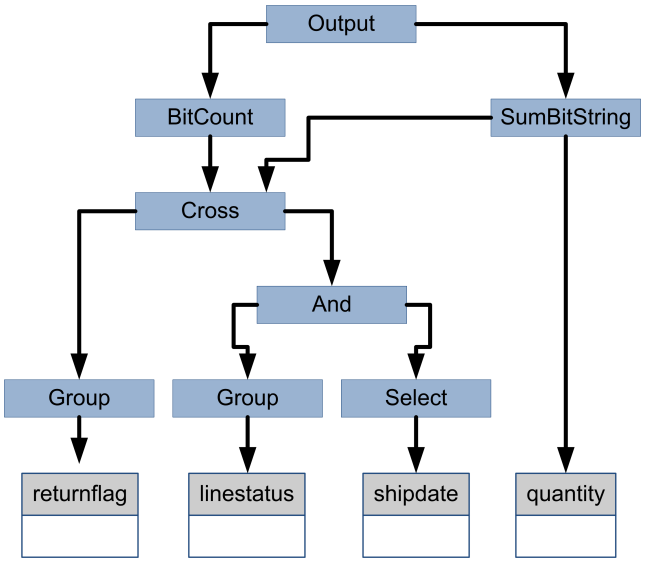


Abbildung 4.6: Ausführungsgraph für die vereinfachte TPC-H-Anfrage Q_1 unter Verwendung von Bit-Strings

Das Ergebnis der Selektion muss anschließend auf die Gruppierung angewendet werden, dazu wird im Beispiel die Schnittmenge aus der Selektion und den Gruppen aus *linestatus* durch die bitweise *Und*-Verknüpfung der Bit-Strings gebildet. Anschließend muss die Gruppierung aus beiden Gruppierungsattributen gebildet werden, dazu wird

[3] außer den Blättern

der *Cross-Operator* verwendet. Zum Schluss werden die Aggregationen mit Hilfe der Operatoren *BitCount* und *SumBitString* gebildet. Die einzelnen Operatoren werden über Operatoren des Hardwaremodells bzw. des physischen Datenmodells dargestellt (s. Tab. 4.3).

Operator im QEP	Operator im Framework
Select	*GetBitStringBySelection* (s. Bsp. 3.11)
Group	*Group* (s. Abschn. 3.3.3.3)
And	*AndOperator* (s. Bsp. 3.10)
Cross	*Cross-Operator* (s. Bsp. 3.14)
BitCount	*BitCount* (s. Abschn. 2.3.2.1)
SumBitString	*AggSumByBitString* (s. Bsp. 3.12)

Tabelle 4.3: Operatoren im QEP und ihre Umsetzung im Framework

4.3.2.2 Joinverarbeitung unter Verwendung von Bit-Strings

Neben der Selektion auf einzelnen Relationen können Bit-Strings auch für die Verarbeitung von komplexen OLAP-Anfragen (s. Abschn. 1.1.1) verwendet werden. Dabei werden zuerst die Prädikate auf den einzelnen Dimensionstabellen ausgewertet und im Anschluss mit Hilfe von Bit-String-Joinindizes die Verbindung zur Faktentabelle vollzogen (s. [103] Kap. 17). Eine sehr ähnliche Art der Ausführung beschreibt Abadi in [1] unter dem Namen „Invisible Join“, nur dass er auf die vordefinierte Joinindizes verzichtet und sie stattdessen zur Laufzeit – in Form von Hashtabellen – erstellt.

Beispiel 4.6: Joinverarbeitung mit Bit-Strings

In diesem Beispiel wird die Ausführung mit Hilfe des im Abschnitt 3.3.1 definierten Domänen-Joinindex veranschaulicht. Dazu wird die TPC-H-Anfrage Q_5 aus Beispiel 3.1 verwendet, die jeweils ein Prädikat auf den Relationen *Region* und *Orders* definiert. Dazu kommt die Gruppierung über den Namen der Relation *Nation* und eine Aggregation auf der *LineItem*-Relation (vgl. Abb. 3.1).

Abbildung 4.7 zeigt den QEP unter der Verwendung des Domänen-Joinindexes (s. Abschn. 3.3.1.5) und des im Framework definierten Operators *Proj* (Def. 3.7). Man kann sehen, dass zuerst die Prädikate mit Hilfe der beiden *Select* Operatoren ausgewertet werden. Im Anschluss wird der Join von *Region* mit *Nation* über die Operatoren *JoinIdx* und *Proj* gebildet. Danach werden auf *Nation* die einzelnen Gruppen mit Hilfe des *Group* Operator gebildet. Dieser Operator erzeugt eine Menge von (Attributwert, Bit-String) Paaren, die dann jeweils den eingekreisten Operatorgraphen durchlaufen. Dabei werden zuerst die Joins auf *Supplier* und *Customer* gebildet, man beachte, dass dieses in einer einzigen Operation geschieht. Weiter werden die jeweiligen Joins bzgl. *LineItem* gebildet, wobei noch die Selektion aus *Orders* mit einbezogen wird. Zum Schluss wird für jede Gruppierung die Aggregation in Form der Summenbildung (*Sum*) gebildet und das erzeugte Paar aus Attributwert der Gruppierung und Aggregationswert in das Ergebnis geschrieben und bzgl. des Aggregationswertes sortiert.

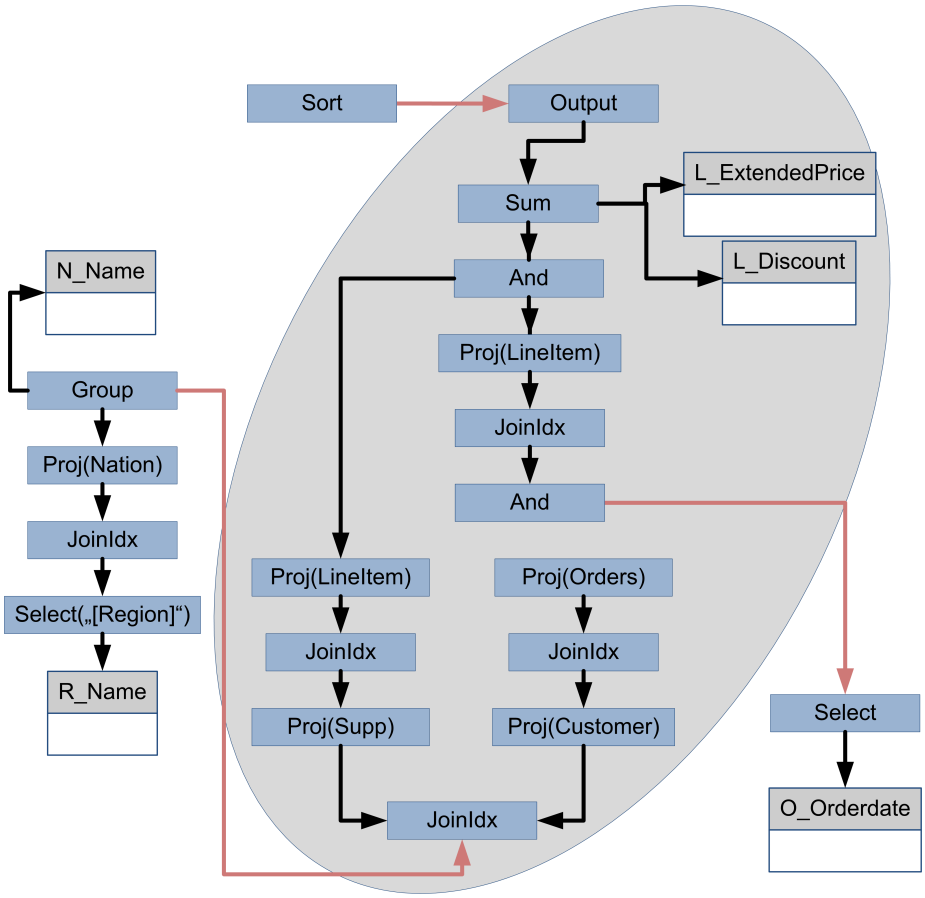


Abbildung 4.7: QEP der TPC-H-Anfrage Q_5

4.3.2.3 Parallele Verarbeitung von Bit-Strings

Im Rahmen der Diplomarbeit von Robert Nagel [127] wurden für Bit-Strings verschiedene Methoden der effizienten parallelen Verarbeitung betrachtet und Messungen angefertigt. Darin werden Bit-String-Operatorbäume betrachtet, deren Operatoren ausschließlich aus bitweisen *Und*- und *Oder*-Operationen bestehen, die mehrere Bit-Strings als Eingabe haben und genau ein Ergebnis-Bit-String erzeugen; die Blätter werden durch Bit-Strings dargestellt (s. Abb. 4.8). Untersucht werden sowohl eine horizontale (datenparallele) als auch eine vertikale (aufgabenparallele) Verarbeitung (s. Abschn. 2.2.2.1).

Die vertikale Verarbeitung ist mit der horizontalen Interoperator-Parallelität (Def. 4.4) zu vergleichen. Jeder Operator des Operatorbaums stellt eine einzelne Task dar, die erst abgearbeitet wird, wenn alle Kind-Tasks beendet sind. Im Rahmen der Arbeit von Robert Nagel wird keine Intraoperator-Parallelität (Def. 4.3) betrachtet.

Bei der horizontalen Ausführung wird der Operatorbaum sequentiell auf partitionierten Bit-Strings ausgewertet. Für die datenparallele Ausführung wird jeder Bit-String in N Partitionen aufgespaltet, so dass im Anschluss maximal N Instanzen des QEPs

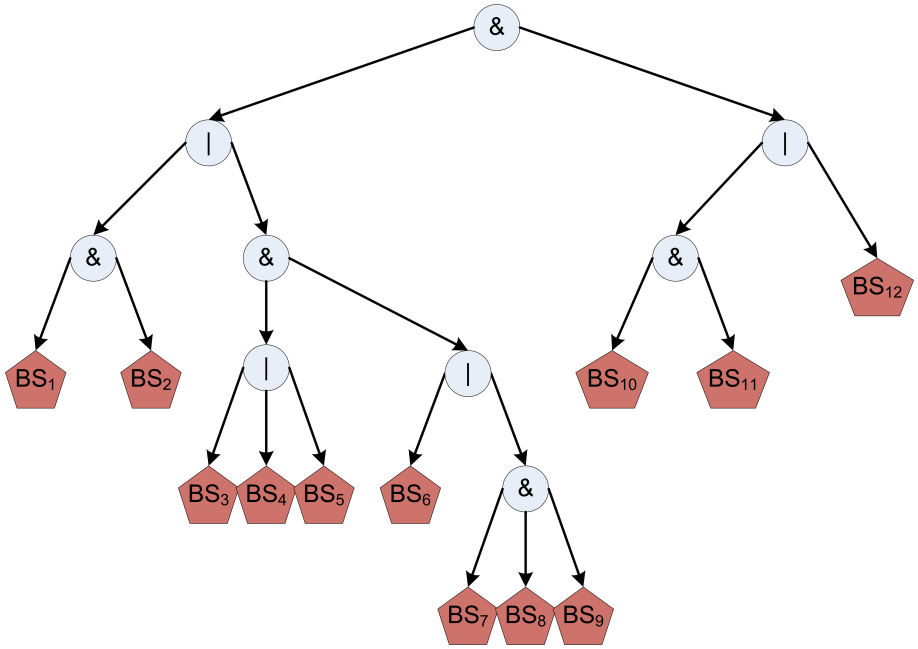


Abbildung 4.8: Operatorbaum für Bit-Strings

parallel ausgeführt werden können; im Abschnitt 5.3.3 folgt ein Modell zur Bestimmung der optimalen Partitionsanzahl N.

Während der bitweisen *Und*- bzw. *Oder*-Berechnung kann eine zusätzliche Leistungssteigerung durch die Nutzung von SIMD-Operationen (Def. 2.8) erreicht werden, indem die Funktion entweder direkt mit SIMD-Befehlen programmiert wird oder durch den Einsatz der automatische Vektorisierung (Def. 2.30). Wie bereits in den Abschnitten 2.2.4.5 und 2.3 dargestellt, sind SIMD-Befehle Rechnerarchitektur abhängig. Das Framework löst diese Abhängigkeit durch die im Hardwaremodell definierten Operatoren (s. Abschn. 2.3) auf und verwendet für die bitweisen *Und*- und *Oder*-Operationen Bibliotheksoperatoren, die automatisch die effizienteste Ausführung auf einer bestimmten Rechnerarchitektur garantieren.

Die Ressourcenverwaltung eines QEPs wird mit Hilfe des Hardwaremodells geplant. Dazu werden über den HW-Graphen (Def. 2.32) mögliche Belegungen gesucht, validiert und bewertet. Bei der horizontalen Ausführung wird dazu je eine Belegung für die gesamte Ausführung pro Partition des Bit-String-Operatorbaums generiert, diese sind strukturell identisch, beanspruchen aber unterschiedliche physische Hardwarebereiche (z.B. Hardware-Threads, Cache). Für die vertikale Ausführung bekommt jeder Operator im Baum seine eigene Belegung zugewiesen, die er dann zur Ausführungszeit beachten muss.

Die Messungen und Auswertungen haben einen deutlichen Vorteil der horizontalen Zerlegung ergeben, da in diesem Fall deutlich mehr unabhängige Tasks entstehen und die vorhandene Parallelität der Hardware besser genutzt werden kann. Ebenfalls kann durch die horizontale Zerlegung der Cache (Def. 2.1) effizienter genutzt werden. Insbesondere bei großen Bit-Strings, die nicht komplett in den Cache passen, kann dieses zu großen

Leistungsgewinnen führen [127].

Im nächsten Abschnitt wird ein im Rahmen dieser Arbeit entwickeltes asynchrones Anfrageausführungsmodell vorgestellt, das für die Ausführung von Bit-String-Operatorbäumen im Framework verwendet wird und alle drei Intraquery-Parallelitäten (Def. 1.7) verwendet. Eine Evaluierung des Modells, sowie der Vergleich zu den Messungen von Robert Nagel, wird im Anschluss gegeben.

4.3.3 Das Asynchrones Anfrageausführungsmodell

Sowohl die datenparallele Ausführung von Bit-String-Operatorbäumen als auch die in Abschnitt 4.3.1 vorgestellte taskbasierte Abarbeitung in MonetDB könnten durch die Verwendung der vertikalen Interoperator-Parallelität einen noch höheren Grad an Parallelität erreichen. Daher war die Verbindung von Interoperator-Parallelität mit Intraoperator-Parallelität in einem Ausführungsmodell ein wichtiges Ziel für diese Arbeit. Nachfolgend wird ein Anfrageausführungsmodell vorgestellt, mit dem sich alle drei Arten von Parallelitäten (Intra-, horizontale Inter- sowie vertikale Interoperator-Parallelität) verbinden lassen und das sich speziell für die Ausführung auf Mehrkern-Rechnerarchitekturen eignet. Zuerst soll anhand der zuvor beschriebenen taskbasierten Ausführung in MonetDB (s. Abschn. 4.3.1) vorgestellt werden, wie die vertikale Interoperator-Parallelität umgesetzt werden kann und welche Vorteile sie bringt.

4.3.3.1 Taskbasierte vertikale Interoperator-Parallelität

Wie bereits mehrfach beschrieben ist eine vertikale Interoperator-Parallelität im Column-at-a-time-Modell nicht möglich. Im Folgenden soll am Beispiel der Anfrage Q_1 (Bsp. 4.1) gezeigt werden, wie eine vertikale Interoperator-Parallelität anhand des Taskkonzepts umgesetzt werden kann und die folgenden Vorteile bietet: (i) Es stehen mehr Tasks zur Verfügung, die parallel bearbeitet werden können. Es wird somit die Parallelität erhöht, was nach dem Amdahlschen Gesetzt (Def. 2.19) eine Erhöhung des möglichen *SpeedUps* zur Folge hat. (ii) Die einzelnen Teilergebnisse der Intraoperator-Parallelität können direkt weiter verarbeitet werden und es entfällt somit der Aufwand für die Reduktion (z.B. das Zusammensetzen der Ergebnis-BAT aus den Einzelergebnissen). Dieses resultiert in einer Reduktion von Speichertransfers und damit einer effizienteren Ausführung (Def. 1.2). (iii) Wird die Granularität für die Tasks günstig gewählt und werden die Tasks in der richtigen Reihenfolge ausgeführt, kann es wie bei der vektorbasierten Verarbeitung in MonetDB/X100 zu Vorteilen in der Ausnutzung von Cachelokalitäten (Def. 2.1) kommen.

Abbildung 4.9 zeigt den Teil des QEPs (vgl. Abb. 4.3), an dem die Verwendung der vertikalen Interoperator-Parallelität verdeutlicht werden soll. Es sind vier parallelisierbare Operatoren zu sehen: Die Selektion (*_9*) und die drei anschließenden *Join*-Operationen (*_16*, *_22*, *_30*). Bei der Selektion wird die Eingangs-BAT (*_1*) in Partitionen zerlegt. Für jede Partition entsteht eine Ergebnis-BAT, die im Head die *OId* des selektierten Tupel und im Tail den jeweiligen Datumswert beinhaltet. Bei der Column-at-a-time-Verarbeitung wird die Selektion parallelisiert und anschließend die Reduktion – Bildung

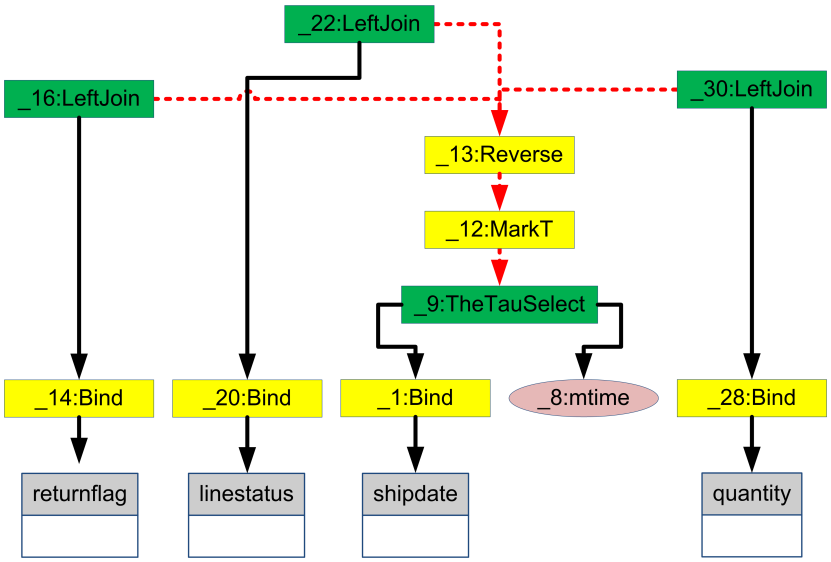


Abbildung 4.9: Ausschnitt des Ausführungsgraphen der vereinfachten TPC-H-Anfrage Q_1

der Gesamtergebnis-BAT – ausgeführt.

Diese BAT durchläuft den Operator *MarkT*, welcher den Tail entfernt und durch neue, in diesem Fall virtuelle *OId*, ersetzt. Die so entstandene BAT wird an den Operator *Reverse* weitergeleitet, welcher einfach nur Head und Tail vertauscht. Es entsteht eine BAT der Form (*VOId*,*OId*), die alle selektierten Tupel aus *LineItem* enthält und ihnen eine neue eindeutige *OIds*, in Form von virtuellen *OIds*, zuweist.

Im Anschluss selektieren die drei Join-Operatoren (*_16*, *_22*, *_30*) dies bezüglich die Attributwerte der Attribute *returnflag*, *linestatus* und *quantity* und weisen ihnen die entsprechenden neuen *OIds* zu. Bei diesen Joins handelt es sich um sogenannte **positionsbezogene Joins** (engl. Positional Joins), diese ermitteln mit Hilfe der Tail-*OId* aus *_13* die Position der Tupel und ihrer Attributwerte in *_14* bzw. *_20* und *_28* und erstellen daraus BATs der Form (*OId*,*AttributType*). Das Ergebnis entspricht dem folgenden Ausdruck der relationalen Algebra (s. Abschn. 3.1):

$$\pi_{\{returnflag,linestatus,quantity\}}(\sigma_{shipdate>=('1998-12-01'-X)}(LineItem))$$

Bei der genauen Betrachtung fällt auf, dass weder für die Join-Operationen noch für *MarkT* oder *Reverse* die komplette BAT notwendig ist; Teilergebnisse können unabhängig von den anderen Ergebnissen weiterverarbeitet werden. Dazu durchlaufen sie unabhängig die beiden Operationen *MarkT* und *Reverse* und die anschließenden Joins. Es ist nur notwendig, dass der *MarkT*-Operator die neuen *OIds* jeweils nur einmal vergibt und z.B. nicht bei jeder Teil-BAT wieder neu bei 0 beginnt.

Diese Eigenschaft kann über verschiedene Verfahren umgesetzt werden, z.B. kann ein gemeinsamer Zähler genutzt werden, der dann synchronisiert werden muss. Bei

einer anderen Variante wird die Partitionsgröße G ausgenutzt, mit der der Selektionsoperator parallelisiert wird. Diese legt gleichzeitig seine maximale Anzahl von Ergebnissen und damit Einträgen in der Teilergebnis-BAT fest (ebenfalls maximal G), damit kann der *MarkT*-Operator für die n-te Partition synchronisationsfrei die *OIds* $nG, ..., (n+1)G-1$ verwenden. Im Allgemeinen bilden sich daraus Löcher zwischen den einzelnen Teilergebnis-*OIds*, diese sind unproblematisch solange nicht das Gesamtergebnis gebildet werden muss und die *OIds* durch virtuelle *OIds* dargestellt werden sollen.

4.3.3.2 Formale Beschreibung des asynchronen Ausführungsmodells

Im Folgenden wird ein datengetriebenes Ausführungsmodell beschrieben, welches mit Hilfe eines Ausführungsgraphens dargestellt wird, bei dem die Knoten die Operatoren repräsentieren und die Kanten den Datenfluss bzw. den Kontrollfluss. Im Modell existieren zwei Arten von Datenflusskanten: (i) Partielle Kanten, die vertikale Interoperator-Parallelität darstellen und mit eine gestrichelte Linie gekennzeichnet sind und (ii) komplette Kanten (durchgezogene Linie), die keine Interoperator-Parallelität zulassen. Dazu kommt eine Kontrollflusskante, die über eine gepunktete Linie markiert wird. Der Datenfluss stellt implizite Abhängigkeiten in der Ausführungsreihenfolge dar.

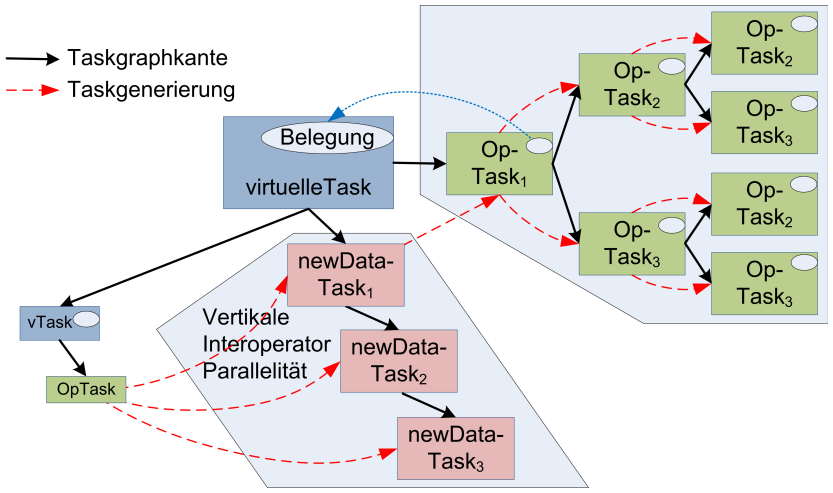


Abbildung 4.10: Taskmodell der Ausführung

Jeder Operator wird über eine **virtuelle Task** dargestellt (s. Abb. 4.10). Die **virtuelle Task** definiert die Bedingungen für die Ausführung (Belegung Def. 2.38), sowie die Ausführungsreihenfolge, also die Interoperator-Parallelität. Der initiale Taskgraph (Def. 2.29) besteht aus virtuellen Tasks und wird anhand der im QEP definierten Abhängigkeiten zwischen den Operatoren aufgebaut.

Die Operatoren selbst sind so aufgebaut, dass sie mit Hilfe der Operationen des Hardwaremodells (s. Abschn. 2.3) und der Operatoren auf dem Datenmodell (s. Abschn. 3.3) parallelisiert werden können. Bei der Ausführung parallelisiert die virtuelle Task die ei-

gentliche Operation und erzeugt dabei Tasks unterhalb von sich selbst (Op-Task$_i$ auf Abb. 4.10). Damit kann die virtuelle Task erst beendet werden, wenn der Operator als ganzes beendet ist. Bei einer kompletten Kante ist damit auch direkt die Ausführungsreihenfolge implizit synchronisiert. In diesem Zustand kann das Modell sowohl Intraoperator-Parallelität als auch horizontale Interoperator-Parallelität darstellen.

Wie bereits beschrieben wird die vertikale Interoperator-Parallelität zwischen zwei Operatoren über eine partielle Kante zwischen ihnen repräsentiert. Bei der Ausführung erzeugt die Intraoperator-Task des datenliefernden Operators eine neue Task unterhalb der virtuellen Task des Konsumenten (newData-Task$_i$ in Abb. 4.10). Diese ist direkt im Kontext des Konsumenten lauffähig und kann die eigentliche Verarbeitung des Konsumenten einleiten.

Beispiel 4.7: Beispielausführung mit Hilfe des asynchronen Ausführungsmodells

```
SELECT SUM(R.V) FROM R,S
  WHERE R.ID = S.FID AND R.A = 'Auto'
    AND R.B = 'Blau' AND S.X = 4711;
```

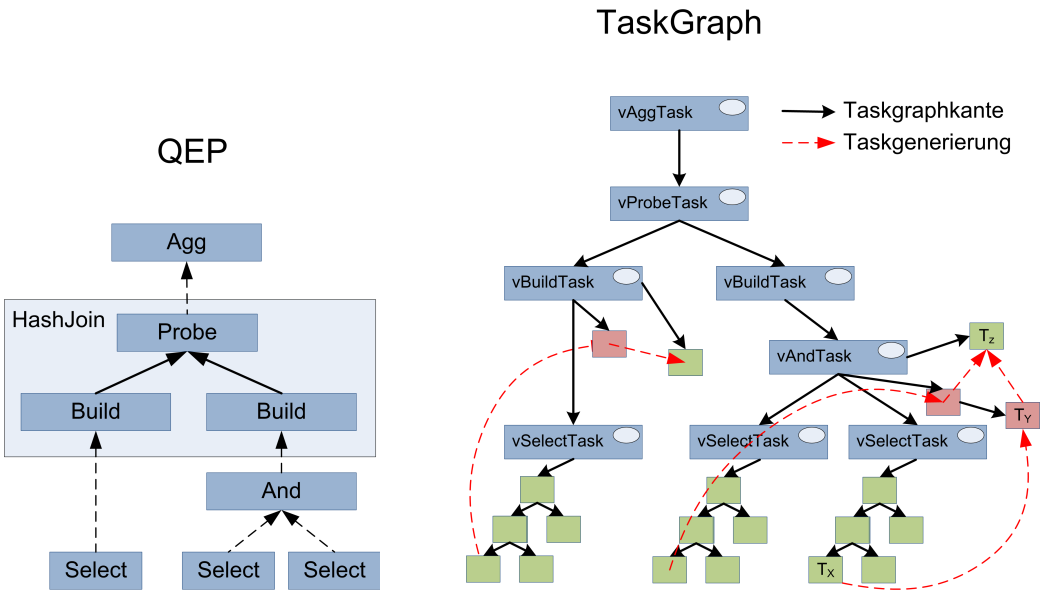
Anhand der obigen Beispielanfrage und des zugehörigen QEPs in Abb. 4.11(a) soll die Ausführung des zuvor definierten Modells verdeutlicht werden. Es sind die drei *Select*-Operatoren, der Aggregationsoperator *Agg*, sowie der Joinoperator – ein symmetrischen Hashjoin – zusehen.

Beim symmetrischen Hashjoin [103] werden zuerst auf den Tupeln beider Eingangsrelationen Hashtabellen erzeugt (**Buildphase**), diese werden im Beispiel durch die beiden (unabhängigen) *Build*-Operatoren realisiert (s. Abb. 4.11(a)). Sobald beide *Buildphasen* beendet sind, geht der symmetrische Hashjoin in seine **Probephase** – durch den *Probe*-Operator dargestellt – über, in der er den Join zwischen den beiden Eingangsrelationen berechnet.

Alle Operatoren bis auf die beiden *Build*-Operatoren haben eine partielle Verbindung zu ihrem Datenkonsumenten im QEP. Somit können die Operatoren *Agg*, *Build* und *And* auch Teilergebnisse ihrer Datenproduzenten verarbeiten und müssen nicht auf das komplette Ergebnis warten, wie im Fall des *Probe*-Operators.

Bei der Abarbeitung wird zuerst der initiale Taskgraph mit den acht virtuellen Tasks erzeugt werden (s. Abb. 4.11(b)). Die drei lauffähigen virtuellen *Select*-Operatoren starten die Ausführung, indem sie die eigentliche Operation parallelisieren und Tasks unterhalb von sich selbst erzeugen (kleine grüne Kästchen unter den virtuellen Tasks). Diese werden dann unter Berücksichtigung der Beschränkungen bzgl. der zugeteilten Ressourcen (Belegungen) ausgeführt.

Beispielhaft wird angenommen, dass die unterste Task (T_x) unterhalb des rechten *Select*-Operators ausgeführt wird und eine Partition der Eingangsrelation R bzgl. des Prädikats $R.B =' Blau'$ ausgewertet hat. Da der *Select*-Operator mit einer partiellen Kante mit dem *And*-Operator verbunden ist, kann am Ende der Berechnung das Teilergebnis weitergereicht werden. Dazu erzeugt die Task T_x eine neue Task T_y unterhalb von der virtuellen *And*-Operator-Task (roter Pfeil) und übergibt darin das Teilergebnis. Die neue Task T_y prüft dann, ob der entsprechende Teil aus dem anderen *Select*-Operator



(a) QEP (b) TaskGraph

Abbildung 4.11: Beispiel für einen QEP und einen Taskgraphen für das Ausführungsmodell

bereits vorhanden ist. Im negativen Fall wird das Ergebnis gespeichert, sonst wird die Operation des *And*-Operators ausgeführt, indem weitere Tasks T_z (grün) unterhalb der virtuellen *And*-Task erzeugt werden, die die eigentliche Intraoperator-Parallelität darstellen.

Die Übergabe-Task T_y kann dabei eine belegungsabhängige Speichertransferoperation ausführen und dazu den dynamischen Laufzeit-Operator *MemTransfer* aus dem Hardwaremodell (s. Abschn. 2.3.2.3) verwenden. Dabei würde das Ergebnis aus T_x von einem NUMA-Knoten (Def. 2.9) zu einem anderen kopiert werden, wenn damit ein Leistungsvorteil erreicht werden kann.

4.3.3.3 Zentraler Koordinationsoperator – WatchDog

Um die Ausführung zu steuern und den initialen Taskgraphen zu erzeugen, existiert für jede Anfrage ein spezieller Koordinationsoperator, genannt **WatchDog**. Neben der Erstellung des initialen Taskgraphen, übernimmt *WatchDog* bei der adaptiven Anfragebearbeitung (s. Abschn. 5.2) eine überwachende und regulierende Rolle. Dazu kann er die einzelnen Taskgraphen bzw. die Menge der noch ausstehenden Tasks betrachten und die Belegungen der virtuellen Tasks ändern, mit dem Ziel, eine optimale Lastbalanzierung zu finden. Die Details dazu werden im Abschnitt 5.3.2 des nächsten Kapitels vorgestellt.

4.3.3.4 Evaluation des taskbasierten, asynchronen Anfrageausführungsmodells

Im letzten Teil folgt eine Evaluation des zuvor beschriebenen Anfragemodells. Dazu wurde zuerst eine auf dem Modells basierende Implementierung angefertigt, die die parallele Verarbeitung von Bit-String-Operatorbäumen (s. Abschn. 4.3.2.3) ermöglicht; im Anschluss folgt eine Evaluierung bzgl. der TPC-H-Anfrage Q_1 in MonetDB unter Verwendung des asynchronen Anfragemodells.

Evaluation von Bit-String-Operatorbäumen

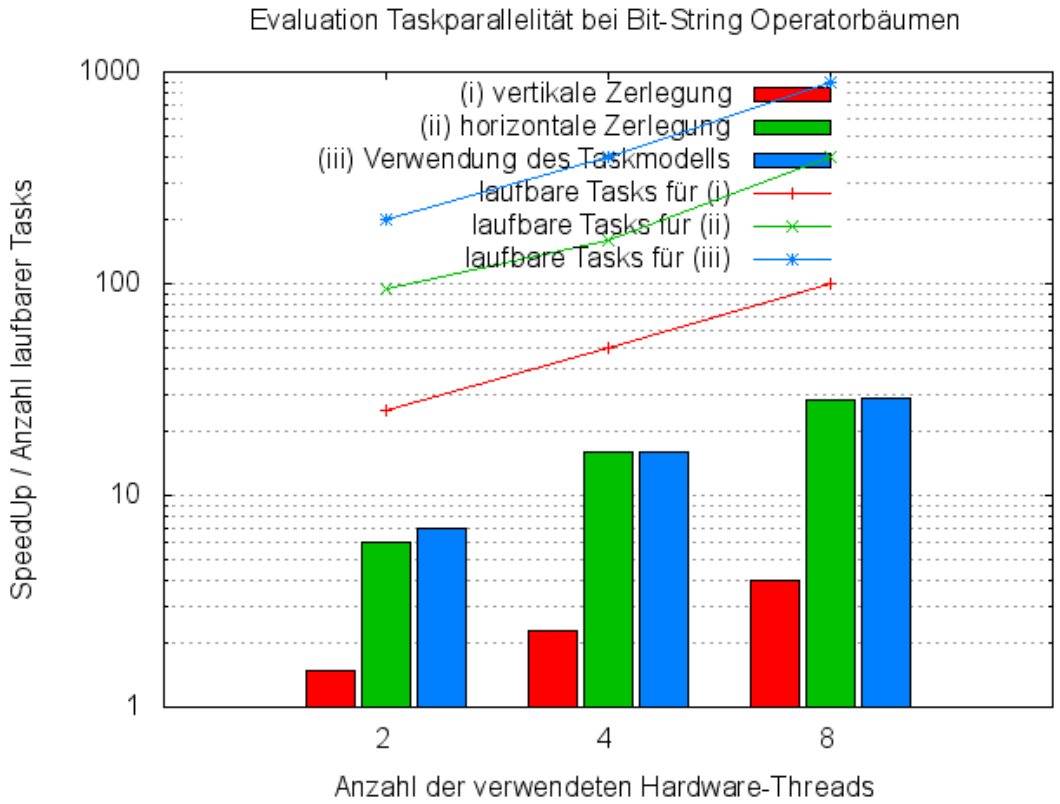


Abbildung 4.12: Evaluation der parallelen Ausführung von Bit-String-Operatorbäumen

Abbildung 4.12 zeigt den Vergleich der Messungen von Robert Nagel bzgl. der vertikalen Zerlegung (rot) und der horizontalen Zerlegung (grün), sowie die Messungen mit Hilfe des asynchronen Anfrageausführungsmodells (blau) bei der Nutzung von maximal acht Hardware-Threads. Dargestellt ist der *SpeedUp* gegenüber der seriellen Verarbeitung des Operatorbaums. Die Verknüpfungsoperationen werden in allen drei Fällen durch abstrakte statische Operatoren (s. Abschn. 3.3.3) umgesetzt.

Zu erkennen ist der deutliche Anstieg des *SpeedUps* – superlinear auf Grund von Cacheeffekten – von der vertikalen zur horizontalen Zerlegung. Der *SpeedUp* des asynchro-

nen Taskmodells kann gegenüber dem der horizontalen Zerlegung nicht weiter erhöht werden, da in beiden Fällen die Ressourcen des Prozessors komplett ausgelastet sind. Im Gegenzug kann man bei der Menge der vorhandenen, lauffähigen Tasks eine deutliche Zunahme sehen, dieser Fakt spricht wiederum für eine gute Skalierung auf einem Rechner mit einer noch höheren Anzahl von Hardware-Threads.

Evaluation TPC-H-Anfrage Q_1

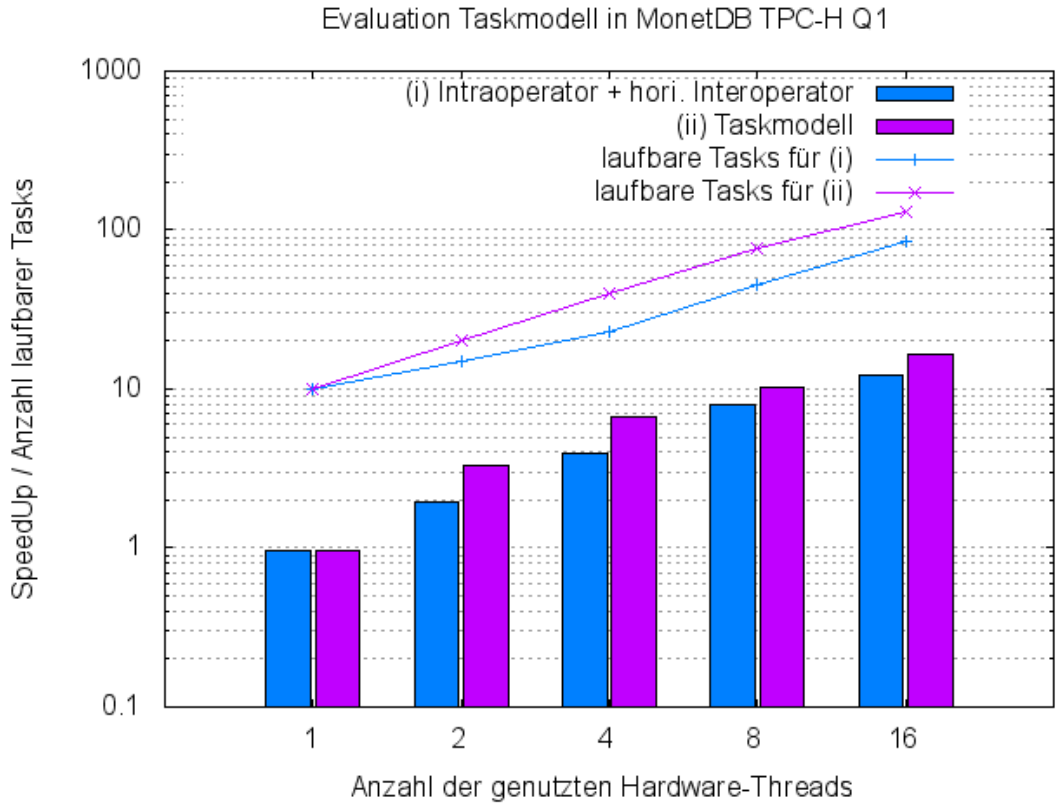


Abbildung 4.13: Auswertung der asynchronen Ausführung der TPC-H-Anfrage Q_1 in MonetDB

Im Rahmen der Arbeit wurde das Konzept des asynchronen Anfrageausführung auch prototypisch in MonetDB umgesetzt. Abbildung 4.13 zeigt die Auswertung bzgl. der TPC-H-Anfrage Q_1 analog zu der Evaluierung in Abschnitt 4.3.1 (s. Abb. 4.5). Dargestellt ist der Vergleich bzgl. einer verschiedenen Anzahl von Threads sowie bzgl. der kombinierten taskbasierten Intra- und horizontalen Interoperator-Parallelität (blau) und der asynchronen Ausführung durch das zuvor beschriebene Anfrageausführungsmodell (violett).

Erkennbar ist im Vergleich zur Kombination aus taskbasierter Intra- und horizontaler Interoperator-Parallelität ein signifikanter Anstieg des *SpeedUp* bei der asynchronen

Ausführung. Er ist auf die bessere Cachenutzung, sowie den Wegfall der Berechnung und der Materialisierung der kompletten Column-at-a-time-Zwischenergebnisse zurückzuführen. Wie in MonetDB/X100 können die Zwischenergebnisse (Vektoren) innerhalb der Planfragmente direkt weiterverarbeitet werden. Damit kommt es zu einer Reduktion von Speichertransfers sowie der Verbesserung des Hit-Miss-Verhältnisses (Def. 2.4).

Zusammenfassung

In Kapitel 4 wurden zunächst verschiedene Anfrageausführungsmodelle vorgestellt und miteinander verglichen. Ehe das Ausführungsmodell von MonetDB um die taskbasierte Intraoperator, sowie horizontale Interoperator-Parallelität erweitert und gezeigt wurde, wie mit Hilfe des Taskkonzepts und des Taskgraphen eine implizite Synchronisation und Lastbalanzierung in der Abarbeitung des QEPs modelliert werden können. Im Anschluss folge die Anfrageausführung unter Verwendung von Bit-Strings für joinfreie Anfrage als auch für Anfragen mit Join; gefolgt von einem neuen datengetriebenen Anfrageausführungsmodell, das auf der Verwendung des Taskkonzepts basiert. Im Modell werden die Operatoren des QEPs durch virtuelle Tasks in einem Taskgraphen dargestellt, der unter Verwendung eines zentralen Koordinationsoperator (*WatchDog*) aufgebaut, ausgeführt und kontrolliert wird.

5 Anfrageoptimierung

„Wer einen Engel sucht und nur auf die Flügel schaut, könnte eine Gans nach Hause bringen.“

Georg Christoph Lichtenberg (1742–1799)

Kapitel 5 ist der Anfrageoptimierung gewidmet. Aufbauend auf der Definition der Anfragebearbeitung (Def. 1.4) werden im ersten Abschnitt grundlegende Konzepte der Anfrageoptimierung betrachtet. Dazu werden zuerst allgemeine Konzepte der Anfrageoptimierung vorgestellt, ehe auf Erweiterungen für die parallele Anfrageoptimierung eingegangen wird. Der zweite Abschnitt stellt das Thema der adaptiven Anfragebearbeitung vor. Im letzten Abschnitt wird der Anfrageoptimierungsteil des Frameworks vorgestellt und verfolgt drei Ziele: (i) Finden einer lokal optimalen und validen Belegung (Def. 2.38) bzgl. des Hardwaremodells (s. Abschn. 2.3.1) für einen gegebenen QEP. Nachfolgend wird auf die Notwendigkeit und die Prüfung der globalen Belegung vorgestellt. (ii) Finden ein möglichst optimalen globalen Belegung für alle aktiven Anfragen. Das letzte Ziel (iii) ist eine adaptive Anfragebearbeitung, wobei sich das Framework auf die dynamische Anpassung der Belegung beschränken wird; während der Anfrageausführung sollen Engpässe und Überlastungen erkannt und durch Änderungen an den Belegungen – neue Ressourcenzuordnung – aufgelöst werden.

5.1 Grundlagen der Anfrageoptimierung

Die Anfrageoptimierung ist der Prozess der Anfragebearbeitung (Def. 1.4), bei dem die interne Repräsentation einer Anfrage in einen möglichst optimalen Anfrageausführungsplan (Def. 1.5) überführt wird. **„Optimal“** bezieht sich in dieser Arbeit auf eine möglichst geringe Ausführungszeit (s. Def. 1.2).

Die Aufgabe der Anfrageoptimierung ist dabei keines Wegs trivial, da auf Grund der Assoziativität und der Kommutativität vieler Operatoren und verschiedener Operatorumsetzungen (z.B. Hashjoin vs. Sort-Merge-Join) viele äquivalente QEPs für ein und dieselbe Anfrage existieren können. Die Phase der Anfrageoptimierung erfolgt im allgemeinen in zwei Stufen (s. Abb. 5.1): Der (i) **logischen Optimierung**, mit Normalisierung und Vereinfachung. Gefolgt von der (ii) **physischen Optimierung** mit Graphtransformationen und Algorithmenauswahl [54, 59, 100, 34].

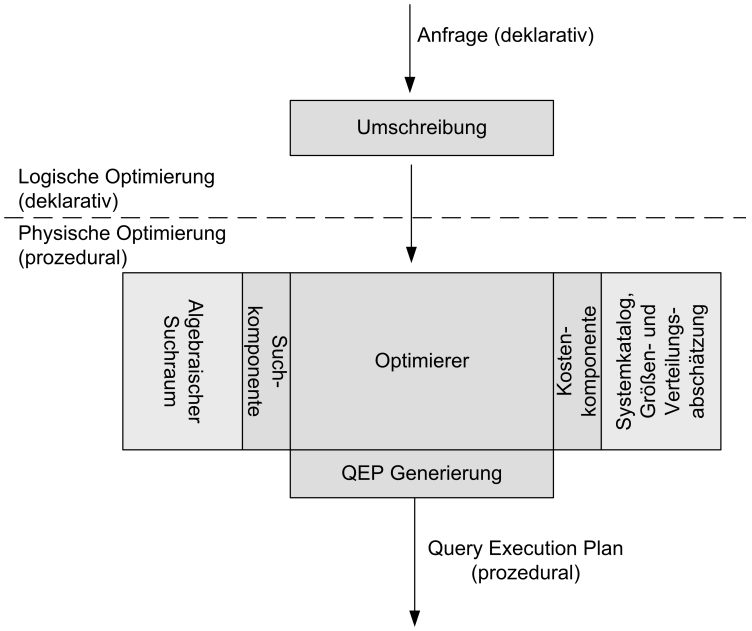


Abbildung 5.1: Prozess der Anfrageoptimierung in RDBMSen

Definition 5.1 (Logische Optimierung) Die **logische Optimierung** normalisiert und vereinfacht den an sie übergebenen Ausdruck. An ihrem Ende steht eine prozedurale, interne Darstellung der Anfrage. □

Definition 5.2 (Physische Optimierung) Die **physischen Optimierung** erzeugt aus einer deklarativen Anfrage einen prozeduralen Ausführungsplan (QEP) und versucht dabei einen QEP zu finden, der bzgl. des zugrunde liegenden Kostenmodells die Gesamtkosten minimiert. In das Kostenmodell fließen für jeden Operator Informationen über Parameter wie z.B. benötigte Betriebsmittel und Eingangsdatengröße ein [61, 53]. Während der physischen Optimierung wird zum einen die Operatorreihenfolge bestimmt, zum anderen wird jedem Operator eine physische Implementierung bzgl. seines Ausführungsalgorithmus zugewiesen. Die physische Optimierung basiert auf der Abschätzung und dem Vergleich der Ausführungskosten von unterschiedlichen QEPs. □

Dreiphasenoptimierung in MonetDB

Eine andere Art der Optimierung beschreiben Kersten et al. in [104] für MonetDB. Dabei schlagen sie eine Anfrageoptimierung in drei Phasen vor: (i) **strategische**, (ii) **taktische** und (iii) **operationale** Optimierung. Die Idee ist, zum einen den extrem großen Suchraum dadurch zu reduzieren, dass bestimmte Entscheidungen erst zu einem späteren Zeitpunkt getroffen werden, zum anderen soll die Verzögerung der Entscheidung auf einen späteren Zeitpunkt einen Informationsgewinn bieten, der dann zu einer „besseren“ bzw. „realitätsnäheren“ Entscheidung führt.

In der ersten Phase, der **strategischen Optimierung**, ist das Hauptziel eine Volumenreduzierung. Dabei werden nur Kosten betrachtet, die unabhängig vom Status des RDBMSs sind (z.B. die Ergebnisgrößen und Sortierreihenfolgen). Der so erzeugte kostenminimale QEP stellt dabei nur teilweise die Ordnung zwischen den Operatoren dar.

In der zweiten Phase versucht die **taktische Optimierung** durch Umschreibung die Ressourcennutzung des QEP aus Phase eins zu minimieren. Phase zwei findet zur Laufzeit statt und versucht auch Überlappungen mit anderen zur Ausführung bereitstehenden bzw. sich bereits in der Ausführung befindenden Anfragen aufzuspüren und auszunutzen. Das Ziel ist die Gesamtausführungszeit aller Anfragen zu minimieren.

Die letzte Phase, die **operationale Optimierung**, findet zur Laufzeit des Operators statt; erst zu diesem Zeitpunkt werden unter Beachtung des aktuelle Zustand des RDBMSs und der Parameter des Operators, die Algorithmen ausgewählt.

Anfrageoptimierung für die parallele Anfrageausführung

Um die einzelnen Parallelitäten im QEP zu finden und zu nutzen, wird in parallelen RDBMSen dem Prozess der Anfrageoptimierung ein dritter und vierter Schritt hinzugefügt, die **Parallelisierung** und das **Scheduling**. Schritt vier erzeugt zunächst durch das Auffinden von Intraquery-Parallelitäten (Def. 1.7) einen parallelen Ausführungsplan (PQEP). Der anschließende Teil des **Schedulings** legt Parameter, wie den Grad der Parallelität, sowie den Ort und die Zeit der Ausführung, fest.

PQEP – Parallel Query Execution Plan

Zwei Phasenverfahren von XPRS

XPRS – eXtended Postgres on Raid and Sprite

XPRS ist ein RDBMS auf Shared-Memory-Architekturen (s. Abschn. 1.1.3) [158]. Es besitzt eine zwei Phasenoptimierung, welche zuerst einen optimalen sequentiellen Plan ermittelt, der dann in der zweiten Phase parallelisiert wird [83], indem er in Fragmente (Def. 4.5) zerlegt wird. Der Algorithmus hat eine ideale Ressourcenauslastung als Ziel; dazu wird je ein CPU-lastiges und ein I/O-lastiges Fragment mit unterschiedlichem Replikationsgrad auf dem System verteilt. Phase zwei wird zur Laufzeit durchgeführt, berücksichtigt aber keinerlei Kosten für die Kommunikation. Eine explizite Modellierung der Kommunikationskosten wird von Hasan in [68] vorgenommen.

Parallele Anfrageoptimierung mit TOPAZ

TOPAZ – TOp-down PArralleliZer

Nippl et al. beschreiben in [128] mit TOPAZ einen fünfstufigen Optimierer, der in der ersten Phase einen sequentiellen QEP erzeugt und dann in den Stufen zwei bis fünf die Parallelisierung vornimmt, wobei alle Arten von Parallelitäten (i.e. Intraoperator, vertikale und horizontale Interoperator) berücksichtigt werden.

5.2 Adaptive Anfragebearbeitung

Bei der Anfragebearbeitung (Def. 1.4), wie sie in dieser Arbeit bisher betrachtet wurde, sind die Anfragepläne (Def. 1.5) und ihre Parameter statisch. Eine einmal getroffene Entscheidung für einen bestimmten Algorithmus wird nicht mehr in Frage gestellt oder verändert. Das gilt sowohl für Entscheidungen, die während der Anfrageoptimierung getroffen wurden, als auch für Entscheidungen, die während der Laufzeit gefällt werden (vgl. 3 Phasenoptimierung Abschn. 5.1). Bestimmte Entscheidungen können allerdings auf Grund von unzureichenden Informationen falsch sein und damit eine optimale Ausführung behindern. Mit der adaptiven Anfragebearbeitung wird versucht, diesem Umstand Sorge zu tragen und Entscheidungen auch nachträglich während der Anfrageausführung zu verändern.

Das Iteratormodell (s. Abschn. 4.2.1) wurde dazu um einen speziellen Operator, dem *Choose* Operator, erweitert. Er kann während der Ausführung zwischen alternativen Teilplänen auswählen [62, 39]. Eine andere Möglichkeit stellt das Konzept des „Eddy“ dar [17]. Mit einem Eddy werden Tupel dynamisch durch verschiedene Operatoren geschleust, ohne eine feste Ausführungsreihenfolge bzgl. der Operatoren definieren zu müssen (s. Abb. 5.2).

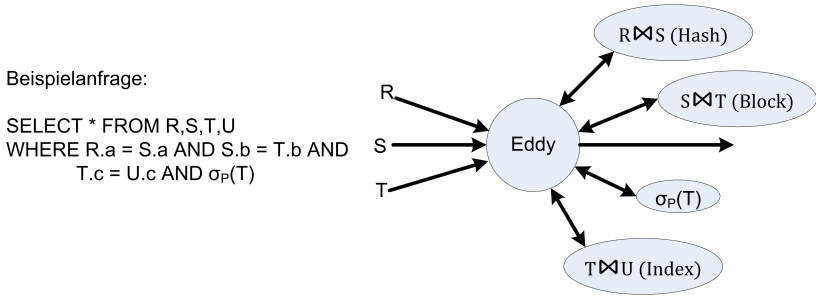


Abbildung 5.2: Eddy: Anwendungsbeispiel vier Wegejoin [44]

Eine weitere Möglichkeit der adaptiven Anfragebearbeitung beschreibt Antoshenkov in [15]. Dabei werden für einen Operator mehrere gleichwertige Realisierungen (z.B. Index-Scan vs. Table-Scan) konkurrierend gestartet. Nach einer bestimmten Zeit wird die schnellere Realisierung weiter verwendet und die anderen abgebrochen.

Einen detaillierten Überblick über das Gebiet der adaptiven Anfragebearbeitung in RDBMSen geben Deshpande et al. in [44] sowie Babu et al. in [19]. Deshpande et al. beschreiben den Prozess der adaptiven Anfragebearbeitung mit einer **Adaptionsschleife** (s. Abb. 5.3). Diese Schleife durchläuft das System wiederholt und beinhaltet vier Teilschritte: (i) Messung: Das adaptive System wird periodisch oder kontinuierlich überwacht. (ii) Analyse: Unter Verwendung der Messungen aus (i), überprüft das System, ob es sich bzgl. eines gesetzten Ziels entwickelt. (iii) Planung: Basierend auf der Analyse, entscheidet das adaptive System, wie das Verhalten geändert werden muss. (iv) Steuerung: Zum Abschluss werden die Entscheidungen aus (iii) im System umgesetzt und das System geht wieder in Zustand (i) über.

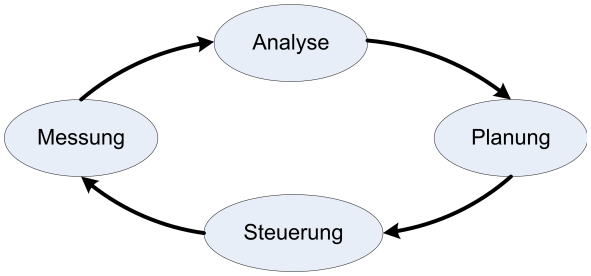


Abbildung 5.3: Adaptionsschleife

Cieslewicz et al. beschreiben in [37] eine Möglichkeit der adaptiven Anfragebearbeitung im Rahmen von Mehrkern-Rechnerarchitekturen. Darin beschreiben sie Möglichkeiten der adaptiven Aggregation auf Mehrkern-Rechnerarchitekturen, indem sie mit Hilfe eines von ihnen vorgestellten Frameworks, während der Anfrageausführung zwischen drei Alternativen der Berechnung wählen können. Diese drei Alternativen beziehen sich auf die Art der Datenaufteilung während der Berechnung; sie unterscheiden: (i) Gemeinsamen Daten für alle Threads, dabei können allerdings Leistungsverluste durch Synchronisation auftreten (s. Abschn. 2.1.1.2). (ii) Verwendung privater Daten; bei diesem Ansatz verwendet jeder Thread seine eigenen Daten bis zum Schluss die Kombination der verschiedenen privaten Ergebnisse berechnet wird. Dieser Ansatz benötigt linear zu der Anzahl von verwendeten Threads mehr Speicher, da jeder Thread seine eigene Kopie der Ergebnisse (z.B. Hashtabelle) besitzt. (iii) Eine Hybrid-Aggregation; sie stellt eine Kombination von (i) und (ii) dar.

5.3 Anfrageoptimierung in Bezug auf Ressourcenverwaltung

Im letzten Teil des Frameworks (s. Abschn. 1.3) soll es um den Prozess der Anfrageoptimierung (s. Abb. 5.4) gehen. Dabei steht besonders die Optimierung der Ressourcenverwaltung bei der parallelen Anfragebearbeitung im Vordergrund. Aufbauend auf dem zwei Phasenverfahren von XPRS (s. Abschn. 5.1) soll im Framework zuerst ein serieller QEP gefunden werden. Anschließend wird der Plan durch blockierende Operatoren in Fragmente (Def. 4.5) zerlegt. Für die Operatoren verschiedener Fragmente werden dann gültige Belegungen (Def. 2.38) gesucht. Dabei soll zuerst nur die Gültigkeit innerhalb der Belegungen eines Fragmentes betrachtet werden (**lokale Belegung**). Im Anschluss wird die Gültigkeit des gesamten Plans als auch einer Menge von laufenden Anfragen betrachtet (**globale Belegung**). Anschließend folgt eine Betrachtung, wie eine adaptive Anfragebearbeitung in Bezug auf die dynamische Änderung von Belegungen möglich ist. Zum Abschluss wird ein Modell vorgestellt, um die Ausführung von Bit-String-Operatorbäumen (Def. 4.6) bzgl. der Ressourcenverwaltung zu optimieren.

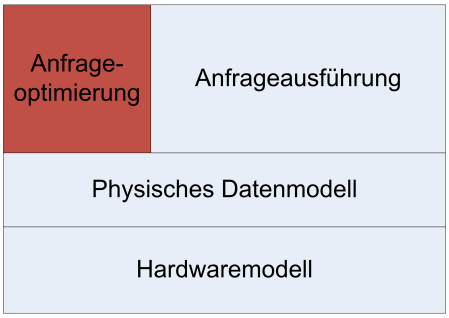


Abbildung 5.4: Framework: Anfrageoptimierung

5.3.1 Statische Belegungserzeugung

Um eine möglichst effiziente (Def. 1.2) Anfrageausführung zu ermöglichen, ist es notwendig, für einen gegebenen seriellen QEP eine optimale parallele Ausführung zu erzeugen, die die gegeben Ressourcen optimal verwendet, ohne sie zu überlasten. Überlastungen können dabei unterschiedliche Ausprägungen haben; in einem Caches tritt eine Überlastung auf, wenn mehr Daten im Cache gehalten werden sollen, als Platz vorhanden ist. Als Folge einer Cacheüberbelastung kommt es zur ungewollten Verdrängung und damit zu einem schlechten Hit-Miss-Verhältnis (Def. 2.4). Überlastungen sollen mit Hilfe von gültigen Belegungen (Def. 2.38) aus dem Hardwaremodell des Frameworks (s. Abschn. 2.3) vermieden oder minimiert werden. Dabei kann das Problem sowohl auf globaler – Gesamtsystemebene – als auch auf lokaler Ebene – QEP oder QEP-Fragmentebene – betrachtet werden.

5.3.1.1 Belegung auf lokaler Ebene

Im Folgenden wird beschrieben, wie jedem Operator eines QEPs oder eines QEP-Fragments eine Belegung zugeordnet wird, ohne dabei einzelne Ressourcen zu überlasten.

Für ein QEP-Fragment F mit n Operatoren soll eine Menge $\mathcal{B}$ mit n Belegungen gefunden werden, so dass für jeden Operator $o \in F$ eine Belegung $b \in \mathcal{B}$ existiert. Weiter muss gelten, dass die Menge von Belegungen $\mathcal{B}$ auf dem HW-Graphen G (Def. 2.32) valide (Def. 2.38) ist, also

$$valid(\mathcal{B}, G) = true$$

gilt. Weiter sollen die geschätzten Kosten c bzgl. einer Kostenfunktion *cost* für das Fragment F mit den Belegungen in $\mathcal{B}$ ein Minimum bilden.

$$\nexists \mathcal{A} | valid(\mathcal{A}, G) = true \wedge cost(\mathcal{A}, F) < cost(\mathcal{B}, F)$$

5.3.1.2 Belegung auf globaler Ebene

Mit der lokalen Ebene wurde versucht, Überbelastungen innerhalb eines QEP-Fragmentes zu vermeiden. Um aber auch Konflikte oder Engpässe auf globaler Ebene zu vermeiden, ist es notwendig, die Belegungsmengen aller vorhandenen QEP-Fragmente zu prüfen. Werden Konflikte zwischen zwei Fragmenten festgestellt, sind zwei Vorgehensweisen möglich: (i) Eine Anpassung der beiden Belegungsmengen der Fragmente, so dass eine valide Gesamtbelegung entsteht. Oder (ii) die Fragmente werden nacheinander ausgeführt; damit werden Konflikte vermieden. Vorstellbar ist auch ein hybrides Konzept mit einem Fragmentscheduler, der zuerst versucht, die Fragmente anzupassen und abhängig von der Kostendifferenz zwischen Belegungsänderung und einer seriellen Ausführung auswählt.

5.3.2 Dynamische Änderung von Belegungen

Das Finden von optimalen, *lokalen* als auch *globalen* Belegungen ist ein rechenintensiver Optimierungsprozess. Außerdem unterliegen den Berechnungen bestimmte Annahmen (z.B. über Datenverteilungen und Ergebnisgrößen), die oft zu falschen Abschätzungen und damit zu suboptimalen Anfrageausführungsplänen führen (s. Abschn. 5.2). Im Framework sollen Techniken der adaptiven Anfragebearbeitung verwendet werden, um Fehler, die während der Planerzeugung gemacht wurden, zur Laufzeit ausbessern zu können. Das Framework wird sich dabei auf die dynamische Zuteilung von Ressourcen bzgl. eines QEPs (Def. 1.5) beschränken und den eigentlichen logischen Ausführungsplan und seine Operatoren konstant halten.

Für die adaptive Anfragebearbeitung spielt der in Abschnitt 4.3.3.3 vorgestellte Koordinationsoperator (*WatchDog*) eine zentrale Rolle. Er überwacht die Ausführung der einzelnen Operatoren seines QEPs in einer Adaptionsschleife (s. Abschn. 5.2) und greift korrigierend in die Ressourcenverwaltung ein. Dieses Vorgehen soll an einem Beispiel verdeutlicht werden.

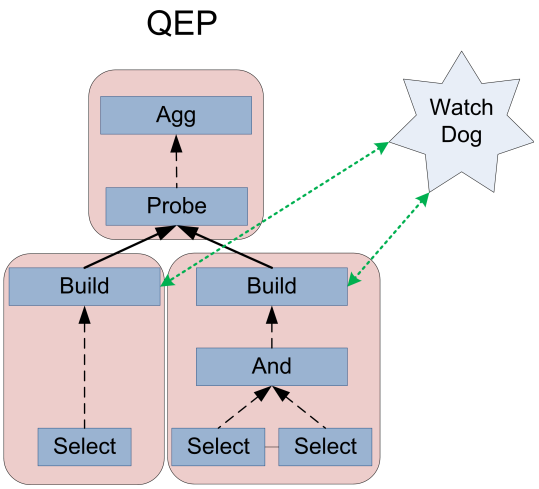


Abbildung 5.5: QEP mit *WatchDog* zu Beispiel 4.7

Beispiel 5.1: Verwendung des *WatchDogs*

Im Beispiel soll die Arbeitsweise des *WatchDogs* am QEP auf Abbildung 5.5 vorgestellt werden. Der QEP bezieht sich auf das Beispiel 4.7, an dem das Anfrageausführungsmodell erläutert wurde.

In diesem Beispiel können die beiden unteren Planfragmente interoperator-parallel (Def. 4.4) ausgeführt werden. Es sei durch die Optimierungsphase (s. Abschn. 5.3.1) für den QEP eine Belegungsmenge erstellt worden, bei der jedem Operator genau zwei Hardware-Threads für die Ausführung zugeteilt wurden.

Die Ausführung beginnt und der *WatchDog* befindet sich bzgl. der Adaptionsschleife (s. Abb. 5.3) im Zustand des Messens. Er überwacht dabei die Anzahl der lauffähigen Tasks für jeden Operator bzw. jeden Thread. Dabei stellt er fest, dass der rechte *Build-Operator* stark überlastet ist, da deutlich mehr Tupel aus dem *And-Operator* kommen als erwartet, da die beiden Selektionen eine starke Korrelation aufweisen und die Gesamtselektivität falsch geschätzt wurde. Für den linken *Build-Operator* kann er dagegen eine zu geringe Auslastung feststellen.

Der *WatchDog* ist nun von der Messphase über die Analyse zur Planung gewechselt. Er kann nun dynamisch reagieren, indem er die Belegungen der beiden *Build* Operatoren ändert. Er nimmt dabei dem linken Operator einen Hardware-Thread weg und fügt ihn dem rechten Operator hinzu. Dazu verändert der *WatchDog* die Belegungen der virtuellen Tasks der beiden Operatoren. Er hat nun die Phase der Steuerung abgeschlossen und geht wieder über in die Messphase.

Der Koordinationsoperator hilft somit, eine bessere und dynamische Ressourcenverteilung zu erreichen und damit ein besseres Leistungsverhalten des RDBMSs.

5.3.3 Ressourcenoptimierung für Bit-String-Operatorbäume

In Abschnitt 4.3.2.3 des Kapitels 4 wurde die Abarbeitung von Bit-String-Operatorbäumen (Def. 4.6) sowohl für die vertikale als auch die horizontale Zerlegung vorgestellt. Bei der horizontalen Zerlegung des Baums ist es wichtig, eine möglichst optimale Zerlegungsgröße zu finden. Dabei sollte die Zerlegungsgröße zum einen möglichst groß sein, um unnötige Zerlegungen und zusätzliches Taskscheduling zu vermeiden. Zum anderen sollte die Berechnung auch den Cache optimal nutzen; berechnete Zwischenergebnisse sollten daher, wenn möglich nicht mehr aus dem Cache verdrängt werden. Für eine optimale Zerlegungsgröße ist somit eine Abschätzung über den Cachebedarf bei der Ausführung der horizontalen Zerlegung notwendig.

Für die Abarbeitung des Operatorbaums wird eine Tiefensuche angenommen, bei der vor der Abarbeitung des Operators o zuerst alle seine Kinder ausgeführt werden müssen (von links nach rechts). Zur Bestimmung des Cachebedarfs für die horizontale Zerlegung des Operatorbaums ist es notwendig, eine Abschätzung über die Anzahl von gleichzeitig im Speicher zu haltenden Zwischenergebnissen zu treffen. Zu diesem Zweck wird der Algorithmus Beispiel 5.2 verwendet. Er durchläuft den Operatorbaum in einer Tiefensuche (Zeilen 4-7) und berechnet dabei schrittweise die Anzahl der im Speicher zu

haltenden Zwischenergebnisse und gibt das Maximum zurück. Neben dem Maximum von Zwischenergebnissen könnte aber auch die durchschnittliche Anzahl von Zwischenergebnissen verwendet werden.

Beispiel 5.2: Bestimmung der Anzahl von Zwischenergebnissen

```
1  int EstMaxTempResults(Operator& o, int& lastCnt, int maxCnt)
2  {
3     int c = o.CntChildren();
4     for(int i=0; i<c; ++i) // left to right order
5     {
6        maxCnt = EstMaxTempResults(o.child[i], lastCnt, maxCnt);
7     }
8     lastCnt += (-c+1);
9     if(maxCnt < lastCnt) return lastCnt;
10    return maxCnt;
11 }
12 // Berechnung für den QEP mit Wurzel r
13 int cnt = 0;
14 int max = EstMaxTempResults(r, cnt, 0);
```

Mit dieser Anzahl lässt sich dann bzgl. einer bestimmten zur Verfügung stehenden Cachegröße $|C|$ die optimale horizontale Zerlegungsanzahl berechnen. Die optimale Segmentgröße f der Bit-Strings bei einer maximalen Anzahl von Zwischenergebnissen z_{max} ist gegeben mit:

$$f = \frac{|C|}{z_{max}}$$

Damit ergibt sich die Zerlegungsanzahl N bei einer Bit-String-Länge von $|bs|$ als:

$$N = \frac{|bs|}{f}$$

Die Verwendung des Maximums für z_{max} gibt dabei die Sicherheit, dass immer alle Zwischenergebnisse gleichzeitig in den Cache C hineinpassen. Bei einem starken Unterschied zwischen dem Maximum und dem Durchschnitt entsteht allerdings eine Überschätzung des Cachebedarfs, was zu einer Erhöhung der Zerlegungsanzahl N führt und Leistungsverluste verursachen kann.

Diese Überlegungen gelten nur für den Fall, dass Zerlegungen immer seriell und nicht parallel abgearbeitet werden und somit keine vertikale Zerlegung stattfindet. Eine optimale Parallelisierung ist aber nur durch die Kombination aus horizontaler und vertikaler Zerlegung erreichbar (vgl. Abschn. 4.3.2.3). Dabei wird der Operatorbaum zuerst in horizontale Fragmente mit der Größe f Byte zerlegt. Anschließend werden die Fragmente in einzelne Tasks aufgeteilt, die dann parallel von P vielen Hardware-Threads bearbeitet werden. Da nun verschiedene Threads an unterschiedlichen Stellen im Operatorbaum Task ausführen, muss die Anzahl von gleichzeitigen Zwischenergebnissen auf eine andere Art und Weise abgeschätzt werden. Im Folgenden wird die Abschätzung zunächst beispielhaft anhand der Abbildung 5.6 vorgestellt, im Anschluss folgt der Algorithmus.

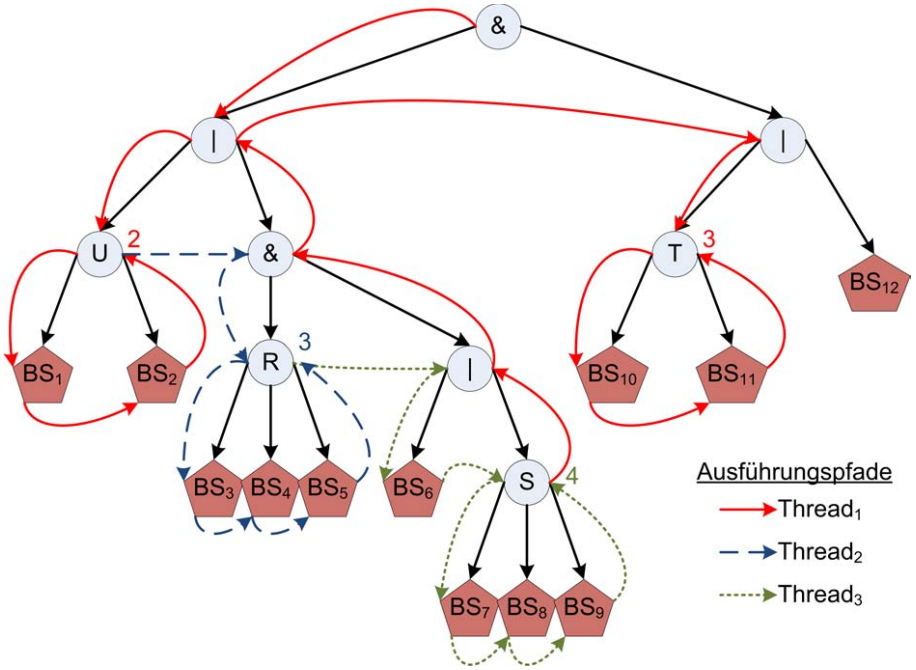


Abbildung 5.6: Operatorbaum für Bit-Strings und seine parallele Abarbeitung

Beispiel 5.3: Maximale Anzahl von Zwischenergebnissen bei paralleler Abarbeitung eines Bit-String-Operatorbaums

Abbildung 5.6 zeigt einen Bit-String-Operatorbaum, bei dem jeder Knoten eine unabhängige Task darstellt; auch für den parallelen Fall gilt, dass Knoten erst dann abgearbeitet können, wenn alle Kindknoten fertig sind.

Der Berechnung der maximalen Anzahl von Zwischenergebnissen z_{max} liegt folgende Beobachtung zu Grunde: z_{max} wird gegenüber der seriellen Abarbeitung nur dann grösser sein, wenn zwei oder mehr Operatorteilbäume gleichzeitig abgearbeitet werden. Die Anzahl der Zwischenergebnisse ist nun genau dann am Größten, wenn alle Threads am Ende eines Operators stehen, bei dem alle Unterbäume bereits abgearbeitet sind. Im Anschluss folgt die eigentliche Operatorausführung, sie hat eine Reduktion der Anzahl von Zwischenergebnissen zur Folge.

In Abbildung 5.6 soll der Operatorbaum mit Hilfe von drei Hardware-Threads abgearbeitet werden. Dazu fängt der Thread $Thread_1$ mit der Ausführung des Operatorbaums an und durchsucht ihn in Tiefensuche nach einer ausführbaren Task und findet Operator U. Vor der Ausführung der Operation von U benötigt $Thread_1$ Platz für zwei Zwischenergebnisse; nach der Ausführung reduziert sich der Platzbedarf wieder auf ein Zwischenergebnis. Auch die Threads $Thread_2$ und $Thread_3$ starten ihrerseits die Ausführung und finden ausführbare Tasks in R und S. $Thread_2$ benötigt dazu Platz für drei Zwischenergebnisse, $Thread_3$ braucht Speicher für vier Zwischenergebnisse.

Die vier Zwischenergebnisse entstehen durch die Abarbeitung des Operatorbaum hinter R. Hierzu muss $Thread_3$ zunächst den Elternoperator „|“ von S ausführen, welcher

das Zwischenergebnis aus BS_6 liefert. $Thread_3$ folgt der Abarbeitung und führt S aus; dabei stehen weitere drei Zwischenergebnisse (BS_7, BS_8 und BS_9). $Thread_3$ hat somit vier Zwischenergebnisse vor der Ausführung von S.

In diesem Zustand haben alle drei Threads gemeinsam eine maximale Anzahl von $z_{max} = 2 + 3 + 4 = 9$ Zwischenergebnissen. Wenn $Thread_1$ den Operator U berechnet hat, führt er die Ausführung weiter (s. Abb. 5.6 rote Pfeile). Im Beispiel sind $Thread_2$ und $Thread_3$ mit R und S noch nicht fertig, daher sind die Elternknoten von U nicht lauffähig und $Thread_1$ beginnt die Ausführung des Elternknoten von T, gefolgt von T selbst. Vor der Ausführung von der Operation von T benötigt $Thread_1$ Speicher für drei Zwischenergebnisse – BS_{10} und BS_{11} sowie für das Ergebnis aus Operator U.

In diesem Zustand benötigen die drei Threads somit Speicher für $z_{max} = 3+3+4 = 10$ Zwischenergebnisse, wir haben somit ein neues Maximum gefunden. Für den gegebenen Bit-String-Operatorbaum ist 10 das Maximum an Zwischenergebnissen, das bei einer parallelen Abarbeitung mit drei Threads entstehen kann.

Der Algorithmus zur Bestimmung der maximalen Anzahl von Zwischenergebnissen für den parallelen Fall soll nachfolgend konstruiert werden. Dazu wird die im Beispiel 5.3 beschriebene parallele Ausführung für eine gegebene Anzahl von Threads P simuliert, so dass die Anzahl der Zwischenergebnisse z_{max} ein Maximum erreicht.

(i) Die Abarbeitung folgt zunächst der seriellen Ausführung bis eine Reduktion ansteht. An diesem Punkt der simulierten Ausführung wird die Anzahl der aktuellen Zwischenergebnisse z_i bestimmt und der Operator der Reduktion wird gespeichert.

(ii) Ausgehend von dem letzten Operator, wird die Abarbeitung des Bit-String-Operatorbaums durch einen weiteren Thread (wie in (i)) simuliert.

(iii) Schritt (ii) wird solange wiederholt bis alle P Threads an einer Reduktion angelangt sind bzw. keine weiteren Operatoren zur Abarbeitung gefunden werden.

(iv) Die maximale Anzahl von Zwischenergebnissen z_{max} muss nun mit der aktuellen Anzahl – $z_{neu} = \sum_{i=1..P} z_i$ – verglichen und eventuell aktualisiert werden.

$$z_{max} = \begin{cases} z_{max} & \text{if } z_{max} >= z_{neu} \\ z_{neu} & \text{sonst} \end{cases}$$

(v) Als nächstes wird die Reduktion bzw. die Ausführung des Operators vorgenommen, der am wenigsten Kinder hat. Dieser Thread simuliert dann die Ausführung wie in (ii) weiter, ohne dabei bereits berechnete Operatoren erneut zu durchlaufen.

Mit Hilfe dieses Algorithmus lässt sich für die Kombination von horizontaler und vertikaler Zerlegung eine optimale Zerlegungsanzahl für die parallele Ausführung von Bit-String-Operatorbäumen berechnen.

Zusammenfassung

In diesem Kapitel wurde der Anfrageoptimierungsteil des Frameworks vorgestellt, dazu wurden zuerst Grundlagen und verschiedene Ansätze der Anfrageoptimierung vorgestellt und anschließend die adaptive Anfragebearbeitung als Mittel zur Fehlerkorrektur bei der QEP-Generierung präsentiert. Im letzten Abschnitt des Kapitel 5 wurde die Verwendung von lokalen und globalen Belegungen motiviert und die adaptive Ressourcenverwaltung mit Hilfe des Koordinationsoperators (*WatchDog*) erklärt. Im letzten Teil folgte ein Modell zur Berechnung der optimalen Zerlegungsanzahl bei der Abarbeitung von Bit-String-Operatorbäumen.

6 Zusammenfassung und Ausblick

„Ein Garten entsteht nicht dadurch, dass man im Schatten sitzt.“

Rudyard Kipling (1865–1936)

Im letzten Kapitel sollen im ersten und zweiten Abschnitt die wichtigen Ergebnisse dieser Arbeit zusammengefasst werden und bzgl. der in Abschnitt 1.3 aus Kapitel 1 gestellten Ziele kritisch beurteilt werden. Der dritte und letzte Abschnitt stellt offene Probleme und Forschungsfragen vor, die im Rahmen dieser Arbeit nicht betrachtet oder gelöst werden konnten.

6.1 Zusammenfassung

Mit dieser Arbeit sollten die Möglichkeiten zur effizienten Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen untersucht werden. Dabei lag der Fokus im Data-Warehouse-Bereich (Def. 1.1) auf Shared-Memory-Architekturen mit Mehrkern-Prozessoren. In diesem Kontext war das Ziel dieser Arbeit die Definition eines Frameworks (s. Abb. 6.1), das diesbezüglich bereits vorhandene und neue Konzepte und Algorithmen miteinander verknüpft. Das Framework ist dazu in vier Bereiche unterteilt, die jeweils in einem eigenen Kapitel vorgestellt wurden.

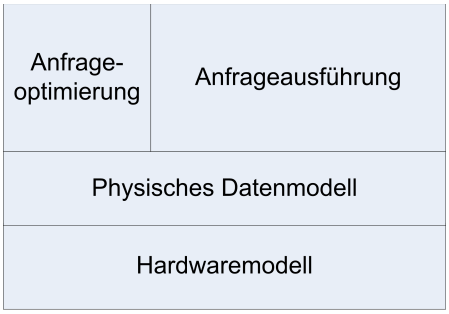


Abbildung 6.1: Framework zur Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen

Die unterste Schicht des Frameworks mit dem Hardwaremodell und seinen Operatoren wurde in Kapitel 2 vorgestellt. Um die zum Teil komplexen Probleme von Mehr-

kern-Rechnerarchitekturen zu verstehen, wurden im ersten Abschnitt sehr ausführlich die Hardwarekomponenten Speicher und Prozessor vorgestellt, gefolgt von verschiedenen Konzepten und Ansätzen zur parallelen Programmierung und einer Einführung in das Konzept der Task-Programmierung in Abschnitt 2. Der letzte Abschnitt stellte den ersten Teil des Frameworks, das Hardwaremodell, vor; darunter wurden die Konzepte des HW-Graphen (Def. 2.32) und seine Belegungen definiert, sowie eine Motivation für verschiedene Klassen und konkrete Beispiele von abstrakten Operatoren gegeben. Den Abschluss stellten zwei Modelle für die Parallelisierung dar.

Kapitel 3 behandelte das physische Datenmodell; es wurde zuerst das Relationenmodell als Modell der konzeptuellen Schicht aus der 3-Schichten-Architektur (Def. 1.3) vorgestellt, und anschließend die physischen Datenmodelle der horizontalen und der vertikalen Partitionierung präsentiert und miteinander verglichen. Im dritten Abschnitt folgte die Erweiterung des Frameworks um das physische Datenmodell. Aufbauend auf dem Konzept der vertikalen Partitionierung wurden zunächst globale Tupelidentifikatoren (gTIDs, Def. 3.6) eingeführt und Anwendungsbeispiel für den Domänen-, Join- und Bereichsindizes gegeben. Weiter wurden Bit-Strings als Realisierung für die Darstellung von gTID-Mengen eingeführt und verschiedene Operatoren auf dem physischen Datenmodell des Frameworks vorgestellt. Der letzte Abschnitt des dritten Kapitels beschäftigte sich mit der Nutzung von SIMD-Operationen für eine effiziente Indexsuche. Dazu wurde ein im Rahmen dieser Arbeit entwickeltes und auf der Segmentierung der Schlüsselwerte basierendes Indexierungsverfahren (Def. 3.14) vorgestellt.

Das Kapitel 4 befasste sich mit der Anfrageausführung und stellte in den ersten beiden Abschnitten Möglichkeiten zur seriellen und parallelen Anfrageausführung vor. Dazu kam eine Bewertung bzgl. der Eignung für die parallele Anfrageausführung auf Mehrkern-Rechnerarchitekturen. Im letzten Abschnitt des vierten Kapitels wurde das Framework um seinen vorletzten Teil, der Anfrageausführung, ergänzt. Dazu wurde zuerst am Beispiel des MonetDB-Systems gezeigt, wie das Taskkonzept bei der parallelen Anfrageausführung helfen kann und anschließend verschiedene Möglichkeiten zur parallelen Ausführung von Bit-String-Operatorbäumen (Def. 4.6) präsentiert. Darauf aufbauend wurde ein Anfrageausführungsmodell entwickelt, das auf dem Taskkonzept basiert und einen maximalen Grad an Parallelität bietet.

Als letzter Teil des Frameworks wurde die Anfrageoptimierung in Kapitel 5 betrachtet. Es wurden zunächst bekannte Konzepte der Anfrageoptimierung im Kontext der seriellen und parallelen Anfragebearbeitung vorgestellt; gefolgt von einem Überblick über das Gebiet die adaptive Anfrageoptimierung. Der letzte Abschnitt präsentiert Ansätze für die Anfrageoptimierung bzgl. der Ressourcenverwaltung auf Mehrkern-Rechnerarchitekturen. Dabei lag der Schwerpunkt auf der Generierung von optimalen und gültigen Belegungen (Def. 2.38) des HW-Graphens (Def. 2.32). Es wurde ferner eine dynamische Ressourcenverwaltung (s. Abschn. 5.3.2) vorgestellt, die die adaptive Anfragebearbeitung erweitert und mit Hilfe eines speziellen Koordinationsoperators (s. Abschn. 4.3.3.3) umgesetzt wird. Den Abschluss bildete ein Modell zur Berechnung der optimalen Zerlegungsanzahl für die horizontale Zerlegung von Bit-String-Operatorbäumen.

Insgesamt hat die Arbeit über die Kapitel 2, 3, Kapitel 4 und Kapitel 5 ein Framework definiert und zusammengesetzt, das eine effiziente Anfragebearbeitung auf Mehrkern-

Rechnerarchitekturen ermöglicht. Dabei wurde Wert daraufgelegt den gesamten Prozess der Anfragebearbeitung inklusive alle seine Bereiche zu betrachten und in der Arbeit und im Framework widerzuspiegeln.

6.2 Beurteilung

Im Folgenden soll die Arbeit, bzgl. des in Abschnitt 1.3 gestellten Ziels, der Schaffung eines Frameworks für eine effiziente Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen, kritisch beurteilt werden. Die Arbeit hatte sich dabei die folgenden sieben Teilziele gesetzt:

- Die Entwicklung eines geeigneten Hardwaremodells, das es erlaubt, die vorhandenen Möglichkeiten und Eigenschaften der Hardware optimal zu nutzen.
- Die Formulierung von speziellen Operatoren als Teil des Hardwaremodells.
- Eine Überprüfung und Erweiterung vorhandener physischer Datenmodelle bzgl. ihrer Eignung für die Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen im Data-Warehouse-Bereich.
- Die Entwicklung eines Indexes, der cacheeffizient ist und die vorhandenen Möglichkeiten der heutigen Mehrkern-Prozessoren optimal nutzen kann.
- Die Entwicklung eines neuen Anfrageausführungsmodells für Mehrkern-Rechnerarchitekturen.
- Die Konzeption der Anfrageoptimierung für Mehrkern-Rechnerarchitekturen bzgl. der Ressourcenverwaltung.
- Eine Realisierung der entwickelten Konzepte in einem funktionsfähigen Prototypen.

Die ersten beiden Teilziele wurden durch die Hardwareschicht in Kapitel 2 erreicht, dabei wurde auf eine vollständige und explizite Auflistung von abstrakten Operatoren verzichtet. Der Grund liegt darin, dass sich eine Vielzahl von Operatoren vorstellen lassen und damit eine Vollständigkeit im Rahmen der Arbeit nicht gewährleistet werden kann. Stattdessen wurden für jede der drei Klassen Beispieloperatoren vorgestellt, die dann eine spätere Wiederverwendung fanden.

Bei der Umsetzung des dritten Teilziels wurde stark auf bereits bekannte Konzepte aufgebaut; dazu wurden verschiedenen Konzepte miteinander verbunden und im Framework untergebracht. Der Fokus lag speziell in der Verwendung von globalen Identifikatoren (gTIDs) und die Realisierung der Repräsentation von gTID-Mengen durch Bit-Strings und deren effiziente Verarbeitung durch parallelisierbare Operatoren.

Teilziel vier wurde durch das neu entwickelte Indexierungsverfahren, den segmentierten Index (Def. 3.14), vervollständigt. Auch Punkt fünf, die Entwicklung eines neuen

Anfrageausführungsmodells für Mehrkern-Architekturen, wurde erreicht. Bei der Konzeption der Anfrageoptimierung, Teilziel sechs, wurden Ansätze präsentiert. Auf eine weitere detailliere Beschreibung der Umsetzung bzw. von Algorithmen lag nicht im Fokus dieser Arbeit.

Das letzte Teilziel war die prototypische Realisierung der entwickelten Konzepte. Speziell die Konzepte des Hardwaremodells konnten nur in Teilen realisiert werden, da auf Grund der Verwendung des TBB-Framework eine explizite Bindung von Tasks an Threads bzw. Hardware-Threads nicht möglich war. Der segmentierte Index konnte dagegen sehr erfolgreich implementiert werden. Auch die verschiedenen Ideen bzw. Konzepte für die Anfrageausführung konnten erfolgreich umgesetzt werden. Auf eine Implementierung der Konzept der Anfrageoptimierung wurde aus zuvor genannten Gründen verzichtet.

Die wesentlichen Ziele für eine effiziente Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen wurden umgesetzt und gezeigt, dass eine effektive Nutzung der vorhandenen Parallelität von Mehrkern-Rechnerarchitekturen im Kontext von RDBMSen möglich ist und sich dabei gute Werte für die Maßzahl *SpeedUp* und *Scale* erreichen lassen. Speziell mit dem segmentierten Index und dem neuen Anfrageausführungsmodell wurden neue Konzepte entwickelt, die die Möglichkeiten von modernen Mehrkern-Rechnerarchitekturen ausnutzen können.

6.3 Ausblick

Im Ausblick sollen noch einmal ungelöste Probleme und weiterführende Arbeiten im Rahmen dieser Arbeit vorgestellt werden.

Bei dem wichtigen Thema der Anfrageoptimierung bzw. der adaptive Anfragebearbeitung konnte die Arbeit Ansätze liefern, bei denen durchaus interessante Möglichkeiten der Weiterführung gegeben sind. Speziell der Bereich der QEP-Erzeugung scheint dabei interessante Forschungsfragen bzgl. der dynamischen Codegenerierung aufzuzeigen [107]. Dabei könnten Probleme, die mit Hilfe der Operatoren aus dem Hardwaremodell gelöst wurden, nochmals vereinfacht werden.

Ein anderes in dieser Arbeit nicht betrachtetes Problem ist die Verarbeitung komprimierter Bit-Strings. Es ist sicher interessant zu untersuchen, welche Komprimierungsmöglichkeiten sich am besten für eine effiziente parallele Verarbeitung von Bit-Strings eignen oder ob zu Gunsten einer höheren Parallelität während der Anfragebearbeitung ganz auf die Komprimierung verzichtet werden sollte oder nur für die Speicherung im Primär- oder Sekundärspeicher zu verwenden ist.

Ein weiterer interessanter Forschungspunkt stellt das Scheduling von Task und die Ressourcenverwaltung dar. Mit dem in dieser Arbeit für die Realisierung verwendete TBB-Framework war keine Möglichkeit gegeben, die Belegungen bzgl. der Hardwareressourcen durchzusetzen. Daher ist eine interessante Frage, ob und wie sich die Belegungen mit Hilfe des ConcRT- oder eines anderen Frameworks ermöglichen lassen.

Akronyme und Abkürzungen

μOps	Micro Operations
ALU	Arithmetic Logical Unit
BAT	Binary Association Table
BF	Breadth-First Search
ccNUMA	Cache Coherent NUMA
CISC	Complex Instruction Set Computer
CMP	Chip-level MultiProcessing
CMT	Chip-level MultiThreading
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CWI	Centrum Wiskunde & Informatica
DAG	Directed Acyclic Graph
DF	Depth-First Search
DRAM	Direct Random Access Memory
DSM	Decomposition Storage Model
DSS	Decision Support System
EX	Instruction Execute
HT	Hyper-Threading
IC	Integrated Circuit
ID	Instruction Decode
IF	Instruction Fetch
ILP	Instruction Level Parallelism
InO	In-Order
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
LDBP	Logical DataBase Processor
LRU	Least Recently Used
MAL	MonetDB Assembler Language
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
Mk-RA	Mehrkern-Rechnerarchitektur
MPMD	Multiple Program Multiple Data
NOP	No OPeration
NSM	N-nary Storage Model
NUMA	Non-Uniform Memory Access
OLAP	OnLine Analytical Processing
OLTP	OnLine Transaction Processing

ONC ..... Open-Next-Close
OoO ..... Out-of-Order
OS ..... Operating System
PAX ..... Partition Attributes Across
PC ..... Program Counter
PDBP ..... Physical DataBase Processor
PPL ..... Parallel Pattern Library
PQEP ..... Parallel Query Execution Plan
QEP ..... Query Execution Plan
QGM ..... Query Graph Model
QPI ..... Quick Path Interconnect
RDBMS ..... Relationales Datenbankmanagementsystem
RISC ..... Reduced Instruction Set Computer
SIMD ..... Single Instruction, Multiple Data
SISD ..... Single Instruction, Single Data
SMP ..... Symmetric Multi Processing
SMT ..... Simultaneous Multi Threading
SPMD ..... Single Program Multiple Data
SQL ..... Structured Query Language
SR ..... Simultaneously Requests
SSE4 ..... Streaming SIMD Extensions 4
TLB ..... Translation Lookaside Buffer
TLS ..... Thread Local Storage
TM ..... Transactional Memory
TOPAZ ..... TOp-down PAralleliZer
TPC ..... Transaction Processing Performance Council
TPC-H ..... TPC ad Hoc
UMS ..... User Mode Scheduling
WB ..... Write Back
XPRS ..... eXtended Postgres on Raid and Sprite

Verzeichnisse

Literaturverzeichnis

[1] Abadi, D. J.: *Query Execution in Column-Oriented Database Systems.* MIT PhD Dissertation, 2008. – PhD Thesis

[2] Abadi, D. J. ; Madden, S. ; Hachem, N. : Column-stores vs. row-stores: how different are they really? In: *SIGMOD Conference*, 2008, S. 967–980

[3] Abrash, M. : *A First Look at the Larrabee New Instructions (LRBni).* `http://www.ddj.com/architect/216402188`, April 2009

[4] Acker, R. ; Roth, C. ; Bayer, R. : Parallel Query Processing in Databases on Multicore Architectures. In: *ICA3PP*, 2008, S. 2–13

[5] Agrawal, S. ; Chaudhuri, S. ; Kollár, L. ; Marathe, A. P. ; Narasayya, V. R. ; Syamala, M. : Database Tuning Advisor for Microsoft SQL Server 2005. In: *VLDB*, 2004, S. 1110–1121

[6] Agrawal, S. ; Narasayya, V. R. ; Yang, B. : Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In: *SIGMOD Conference*, 2004, S. 359–370

[7] Ailamaki, A. ; DeWitt, D. J. ; Hill, M. D. ; Skounakis, M. : Weaving Relations for Cache Performance. In: *VLDB*, 2001, S. 169–180

[8] Ailamaki, A. ; DeWitt, D. J. ; Hill, M. D. ; Wood, D. A.: DBMSs on a Modern Processor: Where Does Time Go? In: *VLDB*, 1999, S. 266–277

[9] Alexandrescu, A. (Hrsg.): *Modern C++ Design, Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001. – ISBN 978–0–20–170431–0

[10] AMD: *Advanced Synchronization Facility.* `http://developer.amd.com/assets/45432-ASF_Spec_2.1.pdf`, März 2009

[11] Amdahl, G. M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference.* ACM, 483–485

[12] Amer-Yahia, S. ; Johnson, T. : Optimizing Queries on Compressed Bitmaps. In: *VLDB*, 2000, S. 329–338

[13] Amer-Yahia, S. ; Johnson, T. : Optimizing Queries on Compressed Bitmaps. In: *The VLDB Journal*, 329-338

[14] Anderson, S. E.: *Bit Twiddling Hacks*. http://graphics.stanford.edu/~seander/bithacks.html, 2005

[15] Antoshenkov, G. : Dynamic Query Optimization in Rdb/VMS. In: *ICDE*, 1993, S. 538–547

[16] Apaydin, T. ; Canahuate, G. ; Ferhatosmanoglu, H. ; Tosun, A. S.: Approximate Encoding for Direct Access and Query Processing over Compressed Bitmaps. In: *VLDB*, 2006, S. 846–857

[17] Avnur, R. ; Hellerstein, J. M.: Eddies: Continuously Adaptive Query Processing. In: *SIGMOD Conference*, 2000, S. 261–272

[18] Azimi, M. ; Cherukuri, N. ; Jayasimh, D. N. ; Kumar, A. ; Kundu, P. ; Park, S. ; Schoinas, I. ; Vaidya, A. S.: Integration Challenges and Tradeoffs for Terascale Architectures. In: *Intel Technology Journal* (2007), August. http://dx.doi.org/10.1535/itj.1103.01. – DOI 10.1535/itj.1103.01

[19] Babu, S. ; Bizarro, P. : Adaptive Query Processing in the Looking Glass. In: *CIDR*, 2005, S. 238–249

[20] Bailey, A. : OpenMP: More than Just Optimizing Loops. (2006)

[21] Barua, R. ; Kranz, D. A. ; Agarwal, A. : Communication-Minimal Partitioning of Parallel Loops and Data Arrays for Cache-Coherent Distributed-Memory Multiprocessors. In: *LCPC*, 1996, S. 350–368

[22] Batten, C. ; Krashinsky, R. ; Gerding, S. ; Asanovic, K. : Cache Refill/Access Decoupling for Vector Machines. In: *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2126–6, S. 331–342

[23] Bayer, R. ; McCreight, E. M.: Organization and Maintenance of Large Ordered Indices. In: *Acta Inf.* 1 (1972), S. 173–189

[24] Bayer, R. ; Unterauer, K. : Prefix B-Trees. In: *ACM Trans. Database Syst.* 2 (1977), Nr. 1, S. 11–26

[25] Beckmann, J. L. ; Halverson, A. ; Krishnamurthy, R. ; Naughton, J. F.: Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. In: *ICDE*, 2006, S. 58

[26] Binstock, A. ; Gerber, R. : *Programming with Hyper-Threading Technology*. http://www.devx.com/assets/intel/9315.pdf, 2003

[27] Boncz, P. A. ; Zukowski, M. ; Nes, N. : MonetDB/X100: Hyper-Pipelining Query Execution. In: *CIDR*, 2005, S. 225–237

[28] Boncz, P. A.: *Monet A Next-Gerneration DBMS Kernel For Query-Intensive Applications*. University Amsterdam, 2002. – PhD Thesis

[29] Burleson, D. : *How and when to use Oracle9i bitmap join indexes*. `http://articles.techrepublic.com.com/5100-10878_11-1051931.html`, 2002

[30] Casazza, J. ; Klein, A. : *Intel QuickPath Architecture on Intel Chip Chat*. `http://video.intel.com/?fr_story=c3a0bedcba8ea7aa19d0a3d7c35a4f339f46c579&rf=bm`, September 2008

[31] Chan, C. Y. ; Ioannidis, Y. E.: Bitmap Index Design and Evaluation. In: *SIGMOD Conference*, 1998, S. 355–366

[32] Chan, C. Y. ; Ioannidis, Y. E.: An Efficient Bitmap Encoding Scheme for Selection Queries. In: *SIGMOD Conference*, 1999, S. 215–226

[33] Chandra, R. (Hrsg.) ; Dagum, L. (Hrsg.) ; Kohr, D. (Hrsg.) ; Maydan, D. (Hrsg.) ; McDonald, J. (Hrsg.) ; Menon, R. (Hrsg.): *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001. – ISBN 1–55860–671–8

[34] Chaudhuri, S. : An Overview of Query Optimization in Relational Systems. In: *PODS*, 1998, S. 34–43

[35] Chaudhuri, S. ; Christensen, E. ; Graefe, G. ; Narasayya, V. R. ; Zwilling, M. J.: Self-Tuning Technology in Microsoft SQL Server. In: *IEEE Data Eng. Bull.* 22 (1999), Nr. 2, S. 20–26

[36] Chynoweth, M. ; Lee, M. R.: *Implementing Scalable Atomic locks for Multi-Core Intel EM64T and IA32 Architectures*. `http://isdlibrary.intel-dispatch.com/isd/85/AtomicLocks_r2.pdf`, 2007

[37] Cieslewicz, J. ; Ross, K. A.: Adaptive Aggregation on Chip Multiprocessors. In: *VLDB*, 2007, S. 339–350

[38] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. In: *Commun. ACM* 13 (1970), Nr. 6, S. 377–387

[39] Cole, R. L. ; Graefe, G. : Optimization of Dynamic Query Evaluation Plans. In: *SIGMOD Conference*, 1994, S. 150–160

[40] Comer, D. : The Ubiquitous B-Tree. In: *ACM Comput. Surv.* 11 (1979), Nr. 2, S. 121–137

[41] Connolly, T. ; Begg, C. : *Database Systems*. Addison Wesley, 2002. – ISBN 9780201708578

[42] Copeland, G. P. ; Khoshafian, S. : A Decomposition Storage Model. In: *SIGMOD Conference*, 1985, S. 268–279

[43] Davis, B. : *Intel Server Update*. http://download.intel.com/pressroom/pdf/nehalem-ex.pdf, May 2009

[44] Deshpande, A. ; Ives, Z. G. ; Raman, V. : Adaptive Query Processing. In: *Foundations and Trends in Databases* 1 (2007), Nr. 1, S. 1–140

[45] DeWitt, D. J. ; Gerber, R. H. ; Graefe, G. ; Heytens, M. L. ; Kumar, K. B. ; Muralikrishna, M. : GAMMA - A High Performance Dataflow Database Machine. In: *VLDB*, 1986, S. 228–237

[46] DeWitt, D. J. ; Ghandeharizadeh, S. ; Schneider, D. A. ; Bricker, A. ; Hsiao, H.-I. ; Rasmussen, R. : The Gamma Database Machine Project. In: *IEEE Trans. Knowl. Data Eng.* 2 (1990), Nr. 1, S. 44–62

[47] DeWitt, D. J. ; Gray, J. : Parallel Database Systems: The Future of High Performance Database Systems. In: *Commun. ACM* 35 (1992), Nr. 6, S. 85–98

[48] Dou, J. ; Cintra, M. H.: A compiler cost model for speculative parallelization. In: *TACO* 4 (2007), Nr. 2

[49] Elmasri, R. ; Navathe, S. B.: *Fundamentals of Database Systems*. Addison-Wesley, 2000

[50] Firasta, N. ; Buxton, M. ; Jinbo, P. ; Nasri, K. ; Kuo, S. : *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency*. http://software.intel.com/file/16820, March 2008

[51] Flynn, M. J.: Toward more efficient computer organizations. In: *AFIPS '71 (Fall): Proceedings of the November 16-18, 1971, fall joint computer conference*. New York, NY, USA : ACM, 1971, S. 1211–1217

[52] Flynn, M. J.: Some Computer Organizations and Their Effectiveness. In: *IEEE Transactions on Computers* (1972), September, Nr. 9, S. 948–960

[53] Freytag, J. C.: A Rule-Based View of Query Optimization. In: *SIGMOD Conference*, 1987, S. 173–180

[54] Freytag, J. C.: The Basic Principles of Query Optimization in Relational Database Management Systems. In: *IFIP Congress*, 1989, S. 801–807

[55] Ghuloum, A. ; Smith, T. ; Wu, G. ; Zhou, X. ; Fang, J. ; Guo, P. ; So, B. ; Rajagopalan, M. ; Chen, Y. ; Chen, B. : *Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture*. http://download.intel.com/technology/itj/2007/v11i4/7-future-proof/7-Future_Proof_Data_Parallel_Algorithms_and_Software.pdf, 2007

[56] Ghuloum, A. ; Sprangle, E. ; Fang, J. ; Wu, G. ; Zhou, X. : *Ct: A Flexible Parallel Programming Model for Tera-scale Architectures*. http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf, 2007

[57] Ghuloum, A. ; Wu, G. ; Zhou, X. ; Fang, J. : *Programming Option Pricing Financial Models with Ct.* http://techresearch.intel.com/userfiles/en-us/File/terascale/Ct-appnote-option-pricing.pdf, 2007

[58] Graefe, G. : Encapsulation of Parallelism in the Volcano Query Processing System. In: *SIGMOD Conference*, 1990, S. 102–111

[59] Graefe, G. : Query Evaluation Techniques for Large Databases. In: *ACM Comput. Surv.* 25 (1993), Nr. 2, S. 73–170

[60] Graefe, G. : B-tree indexes, interpolation search, and skew. In: *DaMoN*, 2006, S. 5

[61] Graefe, G. ; McKenna, W. J.: The Volcano Optimizer Generator: Extensibility and Efficient Search. In: *ICDE*, 1993, S. 209–218

[62] Graefe, G. ; Ward, K. : Dynamic Query Evaluation Plans. In: *SIGMOD Conference*, 1989, S. 358–366

[63] Gustafson, J. L.: Reevaluating Amdahl's Law. In: *Communications of the ACM* 31 (1988), S. 532–533

[64] Haas, L. M. ; Freytag, J. C. ; Lohman, G. M. ; Pirahesh, H. : Extensible Query Processing in Starburst. In: *SIGMOD Conference*, 1989, S. 377–388

[65] Halverson, A. : *Storage and Query Processing Optimizations for Hierarchically-Organized Data.* University of Wisconsin-Madison, 2006. – PhD Thesis

[66] Halverson, A. ; Beckmann, J. L. ; Naughton, J. F. ; DeWitt, D. J.: *A Comparison of C-Store and Row-Store in a Common Framework.* http://pages.cs.wisc.edu/~alanh/tr.pdf, 2006

[67] Harizopoulos, S. ; Ailamaki, A. : StagedDB: Designing Database Servers for Modern Hardware. In: *IEEE Data Eng. Bull.* 28 (2005), Nr. 2, S. 11–16

[68] Hasan, W. : *Optimization of SQL queries for parallel machines.* Stanford University, 1995. – PhD Thesis

[69] Haskell.org: *Haskell.* http://www.haskell.org/, 2010

[70] He, B. ; Luo, Q. : Cache-oblivious databases: Limitations and opportunities. In: *ACM Trans. Database Syst.* 33 (2008), Nr. 2, S. 1–42. http://dx.doi.org/http://doi.acm.org/10.1145/1366102.1366105. – DOI http://doi.acm.org/10.1145/1366102.1366105. – ISSN 0362–5915

[71] Heise Online: *Server Workstation Roadmap AMD.* http://www.heise.de/bilder/107589/0/1, 2008

[72] Heise Online: *AMD demonstiert 12-Kern-Prozessormodul.* http://www.heise.de/newsticker/meldung/136604, April 2009

[73] Heise Online: *Hot Chips: Details zu AMDs Zwölfkener*. http://www.heise.de/newsticker/meldung/144182, August 2009

[74] Heise Online: *Hot Chips: Fujitsu gibt weitere Details zum Sparc64 VIIIfx bekannt*. http://www.heise.de/ct/news/meldung/144278, August 2009

[75] Heise Online: *Hot Chips: Powerr7 -IBMs nächster Server-Chip*. http://www.heise.de/ct/news/meldung/144279, August 2009

[76] Heise Online: *Hot Chips: Suns nächster Server-Prozessor*. http://www.heise.de/ct/news/meldung/144277, August 2009

[77] Heise Online: *SC09: IBM powert mit Power7*. http://www.heise.de/newsticker/meldung/SC09-IBM-powert-mit-Power7-863525.html, November 2009

[78] Herlihy, M. : *Sweet Sixteen: How well is Transactional Memory Aging? – DaMoN2009 Keynote Talk*. http://www.ins.cwi.nl/projects/damon09/damon09_keynote_herlihy.ppt, 2009

[79] Herlihy, M. (Hrsg.) ; Shavit, N. (Hrsg.): *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. – ISBN 978–0–12–370591–4

[80] Hester, P. : *Multi-Core and Beyond: Evolving the x86 Architecture*. http://www.hotchips.org/archives/hc19/3_Tues/HC19.Keynote2/HC19.keynote2.pdf, 2007

[81] Hill, M. D. ; Marty, M. R.: Amdahl's Law in the Multicore Era. In: *IEEE Computer* 41 (2008), Nr. 7, S. 33–38

[82] Hinton, G. ; Sager, D. ; Upton, M. ; Boggs, D. ; Carmean, D. ; Kyker, A. ; Roussel, P. : The Microarchitecture of the Pentium 4 Processor. In: *Intel Technology Journal* (2001), Feburay

[83] Hong, W. ; Stonebraker, M. : Optimization of Parallel Query Execution Plans in XPRS. In: *Distributed and Parallel Databases* 1 (1993), Nr. 1, S. 9–32

[84] Hoskote, Y. ; Vangal, S. ; Dighe, S. ; Borkar, N. ; Borkar, S. : Teraflops Prototype Processor with 80 Cores. (2007)

[85] Howard, P. G.: *Next Generation Intel Microarchitecture Nehalem*. http://www.microway.com/pdfs/microway_nehalem_whitepaper_2009-06.pdf, 2009

[86] Idreos, S. ; Kersten, M. L. ; Manegold, S. : Database Cracking. In: *CIDR*, 2007, S. 68–78

[87] Idreos, S. ; Kersten, M. L. ; Manegold, S. : Self-organizing tuple reconstruction in column-stores. In: *SIGMOD Conference*, 2009, S. 297–308

[88] Inmon, W. H.: The Data Warehouse and Data Mining. In: *Commun. ACM* 39 (1996), Nr. 11, S. 49–50

[89] Inmon, W. H.: *„What is a Data Warehouse?“*. Prism Volumne 1, Number 1, 1995

[90] Inmon, W. H. ; Strauss, D. ; Neushloss, G. : *DW 2.0: The Architecture for the Next Generation of Data Warehousing*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008. – ISBN 0123743192, 9780123743190

[91] Intel: *Micrograph des Pentium 4 Prozessors*. `http://www.cs.umass.edu/~weems/CmpSci635A/Lecture8/p4%283%29.jpg`, 2001

[92] Intel: *Using Spin-Loops on Intel Pentium 4 Processor and intel Xeon Processor*. `http://cache-www.intel.com/cd/00/00/01/76/17689_w_spinlock.pdf`, May 2001

[93] Intel: *Extending the World´s Most Popular Processor Architecture*. `http://download.intel.com/technology/architecture/new-instructions-paper.pdf`, 2006

[94] Intel: 60 Years of continued Transistor shrinkage, Innovation. In: *Intel* (2007)

[95] Intel: *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*. `http://download.intel.com/technology/architecture/new-instructions-paper.pdf`, December 2008

[96] Intel: *Intel Quadcore 45nm die*. `http://download.intel.com/pressroom/kits/45nm/45nm_die_quadcore.jpg`, 2008

[97] Intel: *Intel QuickPath Architecture*. `http://www.intel.com/technology/quickpath/whitepaper.pdf`, 2008

[98] Intel: *Intel Parallel Composer Parallelization Guide*. `http://software.intel.com/en-us/articles/intel-parallel-composer-parallelization-guide`, 2009

[99] Intel: *An Introduction to the Intel QuickPath Interconnect*. `http://www.intel.com/technology/quickpath/introduction.pdf`, January 2009

[100] Ioannidis, Y. E.: Query Optimization. In: *The Computer Science and Engineering Handbook*. 1997, S. 1038–1057

[101] Ivanova, M. ; Kersten, M. L. ; Nes, N. J. ; Goncalves, R. : An architecture for recycling intermediates in a column-store. In: *SIGMOD Conference*, 2009, S. 309–320

[102] Johnson, R. ; Athanassoulis, M. ; Stoica, R. ; Ailamaki, A. : A new look at the roles of spinning and blocking. In: *DaMoN*, 2009, S. 21–26

[103] Kemper, A. ; Eickler, A. : *Datenbanksysteme*. Oldenbourg Verlag München Wien, 2001

[104] Kersten, M. L. ; Manegold, S. ; Boncz, P. A. ; Nes, N. : Macro- and Micro-parallelism in a DBMS. In: *Euro-Par*, 2001, S. 6–15

[105] Khoshafian, S. ; Copeland, G. P. ; Jagodis, T. ; Boral, H. ; Valduriez, P. : A Query Processing Strategy for the Decomposed Storage Model. In: *ICDE*, 1987, S. 636–643

[106] Khronos Group: *Khronos OpenCL API Registry*. `http://www.khronos.org/registry/cl/`, 2010

[107] Krikellas, K. ; Viglas, S. ; Cintra, M. : Generating code for holistic query evaluation. In: *ICDE*, 2010, S. 613–624

[108] Lau, O. : c't extra Programmieren. (2009). ISBN 0–0000–0000–0

[109] Leiserson, C. : *The Cilk Project*. `http://supertech.csail.mit.edu/cilk/`, 2007

[110] Lohman, G. M. ; Lightstone, S. : SMART: Making DB2 (More) Autonomic. In: *VLDB*, 2002, S. 877–879

[111] Lu, H. ; Ooi, B.-C. ; Tan, K.-L. : *Query Processing in Parallel Relational DAtabase Systems*. IEEE Computer Society Press, 1994

[112] Maddox, R. A. ; Singh, G. ; Safranek, R. J.: *A First Look at the Intel QuickPath Interconnect*. `http://www.intel.com/intelpress/articles/A_First_Look_at_the_Intel(r)_QuickPath_Interconnect.pdf`, January 2009

[113] Manegold, S. : *The Calibrator*. `http://homepages.cwi.nl/~manegold/Calibrator/calibrator.shtml`, 2000

[114] Manegold, S. : *Understanding, Modeling, and Improving Main-Memory Database Performance*. University Amsterdam, 2002. – PhD Thesis

[115] Manegold, S. ; Boncz, P. A. ; Kersten, M. L.: Optimizing Main-Memory Join on Modern Hardware. In: *IEEE Trans. Knowl. Data Eng.* 14 (2002), Nr. 4, S. 709–730

[116] Martinez, T. ; Parikh, S. : Understanding Dual-processors, Hyper-Threading Technology, and Multicore Systems. In: *Intel* (2005)

[117] Merkle, B. : *C++0x: Ausblick auf den neuen C++-Standard*. `http://www.heise.de/developer/artikel/Lambda-Funktionen-227388.html`, 2008

[118] Microsoft: *The Manycore Shift*. `http://www.microsoft.com/presspass/events/supercomputing/docs/ManycoreWP.doc`, 2007

[119] Microsoft: *Microsoft WinHEC 2008 Advancing the Platform*. http://download.microsoft.com/download/5/E/6/5E66B27B-988B-4F50-AF3A-C2FF1E62180F/COR-T522_WH08.pptx, 2008

[120] Microsoft: *An Introduction to Native Concurrency in Visual Studio 2010*. http://blogs.msdn.com/nativeconcurrency/archive/2009/01/12/an-introduction-tonative-concurrency-in-visual-studio-2010.aspx, 2009

[121] Microsoft: *Microsoft F#*. http://msdn.microsoft.com/en-us/fsharp/default.aspx, 2010

[122] Microsoft: *User-Mode Scheduling*. http://msdn.microsoft.com/en-us/library/dd627187%28VS.85%29.aspx, 2010

[123] Microsoft: *Streaming SIMD Extensions (SSE)*. http://msdn.microsoft.com/en-US/library/t467de55%28v=vs.80%29.aspx, 2011

[124] Mirman, I. : *Don´t get caught with your multicore pants down!*. http://www.cilk.com/multicore-blog/bid/8097/Don-t-get-caught-with-your-multicore-pants-down, January 2009

[125] Monash, C. : *eBay's two enormous data warehouses*. http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses/, April 2009

[126] Moore, G. E.: Cramming More Components onto Integrated Circuits. In: *Electronics* 38 (1965), April, Nr. 8, 114–117. http://dx.doi.org/10.1109/JPROC.1998.658762. – DOI 10.1109/JPROC.1998.658762

[127] Nagel, R. : *Analyse von parallelen Programmierumgebungen im Kontext von Datenbanksystemen*. Diplomarbeit DBIS-Informatik Humboldt-Universität zu Berlin, 2010

[128] Nippl, C. ; Mitschang, B. : TOPAZ: a Cost-Based, Rule-Driven, Multi-Phase Parallelizer. In: Gupta, A. (Hrsg.) ; Shmueli, O. (Hrsg.) ; Widom, J. (Hrsg.): *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, Morgan Kaufmann, 1998. – ISBN 1–55860–566–5, S. 251–262

[129] O'Neil, P. E.: Model 204 Architecture and Performance. In: *HPTS*, 1987, S. 40–59

[130] O'Neil, P. E. ; Graefe, G. : Multi-Table Joins Through Bitmapped Join Indices. In: *SIGMOD Record* 24 (1995), Nr. 3, S. 8–11

[131] O'Neil, P. E. ; Quass, D. : Improved Query Performance with Variant Indexes. In: *SIGMOD Conference*, 1997, S. 38–49

[132] Pagh, R. ; Satti, S. R.: Secondary indexing in one dimension: beyond b-trees and bitmap indexes. In: *PODS*, 2009, S. 177–186

[133] Panchal, C. : *What is Framework?* http://blog.newtonicaonline.com/what-is-framework/, December 2008

[134] Parys, D. ; Deilmann, D. M.: *Parallele Programmierung mit Visual Studio 2010 und Intel Parallel Studio - Teil 1.* http://www.microsoft.com/germany/msdn/techtalk/videos/library.aspx?id=msdn_de_33301, 2009

[135] Patterson, D. A. ; Hennessy, J. L.: *Computer Organization and Design.* Morgan Kaufmann, 2007

[136] Pawlowski, S. ; Klein, A. : *The Future of Intel Architecture on Intel Chip Chat.* http://video.intel.com/?fr_story=f658b6b0f9ebdc6ca1fc24a8d330339fb2d21cfd&rf=bm, October 2007

[137] Probert, D. : *Windows NT Kernel Going Deep.* http://channel9.msdn.com/shows/Going+Deep/Windows-Part-I-Dave-Probert, March 2005

[138] Probert, D. : *Inside Windows 7 - User Mode Scheduler (UMS).* http://channel9.msdn.com/shows/Going+Deep/Dave-Probert-Inside-Windows-7-User-Mode-Scheduler-UMS/, Feburary 2009

[139] Rao, J. ; Ross, K. A.: Cache Conscious Indexing for Decision-Support in Main Memory. In: Atkinson, M. P. (Hrsg.) ; Orlowska, M. E. (Hrsg.) ; Valduriez, P. (Hrsg.) ; Zdonik, S. B. (Hrsg.) ; Brodie, M. L. (Hrsg.): *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Morgan Kaufmann, 1999. – ISBN 1–55860–615–7, S. 78–89

[140] Rao, J. ; Ross, K. A.: Making B^+-Trees Cache Conscious in Main Memory. In: *SIGMOD Conference*, 2000, S. 475–486

[141] Rao, J. ; Zhang, C. ; Megiddo, N. ; Lohman, G. M.: Automating physical database design in a parallel database. In: *SIGMOD Conference*, 2002, S. 558–569

[142] Rauber, T. (Hrsg.) ; Rünger, G. (Hrsg.): *Multicore: Parallele Programmierung.* Springer, 2008. – ISBN 978–3–540–73113–9

[143] Reinders, J. (Hrsg.): *Intel Thread Building Blocks.* O´Reilly, 2007. – ISBN 0–596–51480–8

[144] Rinfret, D. ; O'Neil, P. E. ; O'Neil, E. J.: Bit-Sliced Index Arithmetic. In: *SIGMOD Conference*, 2001, S. 47–57

[145] Rotem, D. ; Stockinger, K. ; Wu, K. : Optimizing candidate check costs for bitmap indices. In: *CIKM*, 2005, S. 648–655

[146] Saake, G. (Hrsg.) ; Heuer, A. (Hrsg.) ; Sattler, K.-U. (Hrsg.): *Datenbanken: Implementierungstechniken.* mitp, 2005. – ISBN 3–8266–1438–0

[147] Sattler, K.-U. : Column Stores. In: *Datenbank-Spektrum* 30 (2009), S. 39

[148] Schlegel, B. ; Gemulla, R. ; Lehner, W. : k-ary search on modern processors. In: *DaMoN*, 2009, S. 52–60

[149] Seiler, L. ; Carmean, D. ; Sprangle, E. ; Forsyth, T. ; Abrash, M. ; Dubey, P. ; Junkins, S. ; Lake, A. ; Sugerman, J. ; Cavin, R. ; Espasa, R. ; Grochowski, E. ; Juan, T. ; Hanrahan, P. : Larrabee: a many-core x86 architecture for visual computing. In: *ACM Trans. Graph.* 27 (2008), Nr. 3

[150] Silberschatz, A. ; Korth, H. F. ; Sudarshan, S. : *Database System Concepts Third Edition.* McGraw Hill, 1997

[151] Singhal, R. ; Klein, A. : *Nehalem Microarchitecture on Intel Chip Chat.* `http://video.intel.com/?fr_story=3df013ec9784da4a6c026476b25d78cfd467be20&rf=bm`, June 2008

[152] Spracklen, L. ; Abraham, S. G.: Chip multithreading: opportunities and challenges. In: *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, 248–252

[153] Stewart, D. : *Don´t Thread! The End of the Free Ride.* `http://software.intel.com/en-us/blogs/2006/11/15/dont-thread-the-end-of-the-free-ride/`, November 2006

[154] Stockinger, K. : Design and Implementation of Bitmap Indices for Scientific Data. In: *IDEAS*, 2001, S. 47–57

[155] Stonebraker, M. : The Case for Shared Nothing. In: *IEEE Database Eng. Bull.* 9 (1986), Nr. 1, S. 4–9

[156] Stonebraker, M. ; Abadi, D. J. ; Batkin, A. ; Chen, X. ; Cherniack, M. ; Ferreira, M. ; Lau, E. ; Lin, A. ; Madden, S. ; O'Neil, E. J. ; O'Neil, P. E. ; Rasin, A. ; Tran, N. ; Zdonik, S. B.: C-Store: A Column-oriented DBMS. In: *VLDB*, 2005, S. 553–564

[157] Stonebraker, M. (Hrsg.) ; Hellerstein, J. M. (Hrsg.): *Readings in Database Systems, Fourth Edition.* Morgan Kaufmann, 2005. – ISBN 0–26269–314–3

[158] Stonebraker, M. ; Katz, R. H. ; Patterson, D. A. ; Ousterhout, J. K.: The Design of XPRS. In: *VLDB*, 1988, S. 318–330

[159] Sutter, H. : *Software and the Concurrency Revolution.* `http://irbseminars.intel-research.net/HerbSutter.pdf`, 2007

[160] Sybase: *Sybase IQ*. http://www.sybase.com/products/datawarehousing/sybaseiq, 2010

[161] Taniar, D. ; Leung, C. ; Rahayu, W. ; Goel, S. : *High Performance Parallel Database Processing and Grid Databases*. John Wiley & Sons. Inc., 2008

[162] (TPC), T. P. P. C.: *TPC BenchmarkTM H (Decission Support)*. www.tpc.org. Version: 2008

[163] Tremblay, M. ; Chaudhry, S. : A Thrid-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. (2008)

[164] Walter, J. : *Mikrocomputertechnik mit der 8051-Controller-Familie*. Springer, 2007

[165] Weikum, G. ; König, A. C. ; Kraiss, A. ; Sinnwell, M. : Towards Self-Tuning Memory Management for Data Servers. In: *IEEE Data Eng. Bull.* 22 (1999), Nr. 2, S. 3–11

[166] Wikipedia: *Thread computerscience*. http://en.wikipedia.org/wiki/Thread_(computer_science), 2009

[167] Wikipedia: *C++0x*. http://en.wikipedia.org/wiki/C%2B%2B0x, 2010

[168] Wikipedia: *Surrogate Key*. http://en.wikipedia.org/wiki/Surrogate_key, 2010

[169] Winter Corporation: *Winter TopTen Program*. http://www.wintercorp.com/VLDB/2005_TopTen_Survey/TopTenProgram.html, 2005

[170] Winter Corporation: *The Vertica Analytic Database*. http://www.wintercorp.com/WhitePapers/The%20Vertica%20Analytic%20Database%20-%20WinterCorp%20Executive%20Report%20-%202008.pdf, 2008

[171] Wu, K. ; Otoo, E. J. ; Shoshani, A. : On the performance of bitmap indices for high cardinality attributes. In: *VLDB*, 2004, S. 24–35

[172] Wu, M.-C. : Query Optimization for Selections Using Bitmaps. In: *SIGMOD Conference*, 1999, S. 227–238

[173] Xu, L. ; Klein, A. : *Intel QuickPath Architecture on Intel Chip Chat*. http://video.intel.com/?fr_story=43c48a65096c82b798c2c721ee3636ab554f82c7&rf=bm, June 2009

[174] Zhou, J. ; Ross, K. A.: Implementing database operations using SIMD instructions. In: *SIGMOD Conference*, 2002, S. 145–156

[175] Zukowski, M. : *Parallel Query Execution in Monet on SMP Machines*. 2002

[176] Zukowski, M. ; Nes, N. ; Boncz, P. A.: DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In: *DaMoN*, 2008, S. 47–54

Definitionsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Beispielverzeichnis

Lebenslauf

Mein Lebenslauf wird aus Datenschutzgründen in der elektronischen Version meiner Arbeit nicht mit veröffentlicht.

Selbständigkeitserklärung

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift „Anfragebearbeitung auf Mehrkern-Rechnerarchitekturen“ selbständig und ohne unerlaubte Hilfe angefertigt habe;
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze, und
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist, gemäß amtliches Mitteilungsblatt Nr. 34/2006.

Berlin, den 19. Mai 2011 Frank Huber